AD A045722

Stanford Artificial Intelligence Laboratory
Memo AIM-296

Computer Science Department
Report No. STAN-CS-77-592

December 1976

159 p.

A PRACTICAL FORMAL SEMANTIC DEFINITION AND VERIFICATION SYSTEM FOR TYPED LISP

by

Robert Cartwright, Jr.

Doctoral thesis,

STAN-CS-77-592,
AIM-296

MDA903-76-C-0206,
ARPA Order-2494

Research sponsored by

National Science Foundation
and
Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

AD No.
DDC FILE COPY

DDC
OCT 28 1977

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

094 120

Stanford Artificial Intelligence Laboratory
Memo AIM-296

December 1976

Computer Science Department
Report No. STAN-CS-77-592

A PRACTICAL FORMAL SEMANTIC DEFINITION AND VERIFICATION SYSTEM FOR TYPED LISP

by

Robert Cartwright, Jr.

ABSTRACT

Despite the fact that computer scientists have developed a variety of formal methods for proving computer programs correct, the formal verification of a non-trivial program is still a formidable task. Moreover, the notion of proof is so imprecise in most existing verification systems, that the validity of the proofs generated is open to question. With an aim toward rectifying these problems, the research discussed in this dissertation attempts to accomplish the following objectives:

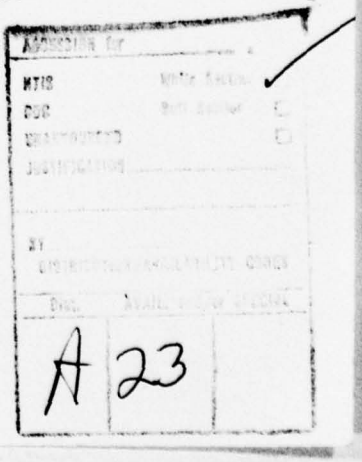
1. To develop a programming language which is sufficiently powerful to express many interesting algorithms clearly and succintly, yet simple enough to have a tractable formal semantic definition.

2. To completely specify both proof theoretic and model theoretic formal semantics for this language using the simplest possible abstractions.

3. To develop an interactive program verification system for the language which automatically performs as many of the straightforward steps in a verification as possible. [continued next page]

This research was supported by the National Science Foundation under Contract NSF MCS76-00327 and Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or any agency of the U. S. Government.

Available from University Microfilm, P. O. Box 1346, Ann Arbor, Michigan 48106.



The first part of the dissertation decribes the motivation for creating TYPED LISP, a variant of PURE LISP including a flexible data type definition facility allowing the programmer to create arbitrary recursive types. It is argued that a powerful data type definition facility not only simplifies the task of writing programs, but reduces the complexity of the complementary task of verifying those programs.

The second part of the thesis formally defines the semantics of TYPED LISP. Every function symbol defined in a program P is identified with a function symbol in a first order predicate calculus language Lp. Both a standard model Mp and a natural deduction system Np are defined for the language Lp. In the standard model, each function symbol is interpreted by the least call-by-value fixed-point of its defining equation. An informal meta-mathematical proof of the consistency of the model Mp and the deductive system Np is given.

The final part of the dissertation describes an interactive verification system implementing the natural deduction system Np.

The verification system includes:

1. A subgoaler which applies rules specified by the user to reduce the proof of the current goal (or theorem) to the proof of one or more subgoals.

2. A powerful simplifier which automatically proves many non-trivial goals by utilizing user-supplied lemmas as well as the rules of Np.

With a modest amount of user guidance, the verification system has proved a number of interesting, non-trivial theorems including the total correctness of an algorithm which sorts by successive merging, the total correctness of the McCarthy-Painter compiler for expressions, the termination of a unification algorithm and the equivalence of an iterative algorithm and a recursive algorithm for counting the leafs of a tree. Several of these proofs are included in an appendix.

This thesis was submitted to the Department of Computer Sc[illegible]d the Committee on Graduate Studies of Stanford University in partial fulfillment of the r[illegible]irements for the degree of Doctor of Philosophy.

PREFACE

This dissertation is not aimed at the casual reader. A shorter, less rigorous, and much more readable account of the same research appears in the *Proceedings of the Third International Colloquium on Automata, Languages, and Programming*, (1976) Edinburgh Press, Edinburgh, under the title "User-Defined Data Types as an Aid to Verifying LISP Programs."

I would like to thank my advisor David Luckham for his patient guidance and encouragement, and my colleagues Derek Oppen, Nicholas Littlestone, and Richard Weyhrauch for their helpful advice and criticism. All the mistakes are mine.

TABLE OF CONTENTS

Chapter 1. INTRODUCTION 1
Section 1. Research Objective 1
Section 2. Previous Work 2
Section 3. Motivation for Creating TYPED LISP 4

Chapter 2. TYPED LISP 8
Section 1. Informal Description of TYPED LISP 8
1. Data Type Definitions 8
2. Function Definitions 10
3. Expressions 11
Section 2. Syntax of TYPED LISP 13
Section 3. Semantics of TYPED LISP 20
1. Assertion Language Syntax 20
2. Assertion Language Semantics 23
3. The Semantics of Program Composition 31

Chapter 3. A FORMAL DEDUCTION SYSTEM FOR TYPED LISP 33
Section 1. Introduction 33
Section 2. A_P: An Axiom System for the Standard Model 33
Section 3. Completeness of the Axiom System A_P 41

Chapter 4. A NATURAL DEDUCTION SYSTEM FOR TYPED LISP 43
Section 1. Introduction 43
Section 2. Expression Simplification Rules 44
Section 3. Formula Simplification Rules 47
Section 4. Goal Simplification Rules 49
Section 5. General Proof Rules 49

Chapter 5. THE IMPLEMENTED VERIFICATION SYSTEM 54
Section 1. Introduction 54
Section 2. Structure of the Verifier 54
Section 3. Demonstration of the Verifier 56
Section 4. Capabilities of the Verification System 70

Chapter 6. FURTHER WORK 71
Section 1. Improving the verification system 71
Section 2. Proving Theorems About Partial Functions 71
Section 3. Extending TYPED LISP 72

REFERENCES 73

APPENDIX 1. SAMPLE PROOFS 74
Section 1. Example 1: Iterative REVERSE 74
Section 2. Example 2: Total correctness of FLATTEN 77
Section 3. Example 3: Total Correctness of Sorting by Merging 83
Section 4. Example 4: McCarthy-Painter Compiler for Expressions 120

APPENDIX 2. TLV USER'S MANUAL 137
Section 1. TLV Conventions 137
Section 2. TLV User Commands 140
Section 3. Running TLV 142
Section 4. TYPED LISP Syntax Error Messages 144
1. Minor Errors 144
2. Major Errors 144
Section 5. TYPED LISP Verifier Command Errors 147

APPENDIX 3. CALL-BY-VALUE FIXED-POINTS 150

CHAPTER 1

INTRODUCTION

1.1 Research Objective

During the past fifteen years, computer scientists have developed a variety of techniques for proving programs correct. Unfortunately, none of these methods have reached the stage where they are practical programming tools. The verification of typical production programs is still far beyond the capability of existing verification systems.

Program verification researchers have frequently ignored practical considerations. Many verification methods employ complex, counter-intuitive formalisms which confuse most computer scientists and totally mystify ordinary programmers. Proofs in these systems tend to be unnatural and very difficult to understand. There is little prospect that they will ever be widely used in practical verification systems. Still other approaches to verification try to reduce the correctness of a program to some other logical problem better suited to mechanization (such as the validity of a single predicate calculus formula). Unfortunately, mechanically "solving" the transformed problem for a non-trivial program is an unfeasibly huge computation (infinite if the program is incorrect). Moreover, the transformed problem often is so unintelligible to the programmer that it is virtually impossible for him to solve--even with the aid of an interactive theorem prover.

Another distressing trend in program verification research has been a careless disregard for firm logical foundations. The notion of proof is so vaguely treated in many verification systems that the "correctness proofs" generated by the systems are of dubious value. Proving a statement using such a system provides little assurance that the statement is true. "Verifying" a program only reduces the correctness of the program to the correctness of the verification system involved (including both the methodology and the proof-checking programs). Computer scientists have been very lax in scrutinizing proposed verification methods for logical flaws.

In the machine implementation of verification systems, there has been far too much emphasis placed on total automation. Most implemented verification systems are almost completely automatic, but none of these automatic systems can verify more that a very limited set of simple programs. Despite intense research efforts, the performance of theorem proving programs still does not approach the level required for automatic verification of non-trivial

programs. Furthermore, there is no existing methodology which suggests that sufficiently powerful theorem provers are on the horizon. Completely automatic verification fails to exploit the programmer's intuitive understanding of the programs he creates. Interactive verification controlled by the programmer is a much more promising approach which has not received sufficient attention from researchers in the field.

With these criticisms in mind, my goal has been to develop logically sound, interactive verification methods which show promise of having practical applications. In order for a programmer to guide a verifier through a proof of program correctness, he must understand the proof steps generated by the verifier. Consequently, the formal system used by an interactive verifier should be as intuitively transparent and natural as possible. For this reason, I selected first order predicate calculus with equality as the basis for my formal system. Proofs in well-designed predicate calculus natural deduction systems (originally developed by Gentzen) closely correspond to their informal counterparts. Furthermore, first order predicate calculus is a very well understood formal system which has received near universal acceptance among mathematicians as the appropriate system for formalizing mathematical theories.

Programming languages vary widely in their suitability for verification. Ideally a programming language should permit the programmer to directly formalize the simplest, most abstract description of the algorithm he wishes to implement. On the other hand the language should have a brief, tractable formal definition so that we can be confident we have correctly defined its semantics. Consequently, I chose PURE LISP as the basis for my verification system's programming language. PURE LISP is sufficiently powerful to concisely express many complex algorithms, yet it has a simple formal definition. My initial idea was to develop a first-order theory of LISP S-expressions analogous to Peano's axioms for the natural numbers, and then to define the semantics of LISP programs by treating each LISP function as a new primitive function satisfying its defining equation. In other words, for each function definition $f(x_1,...,x_n) = \tau(x_1,...,x_n)$ in a LISP program, where $\tau(x_1,...,x_n)$ is a LISP expression, the axiom $f(x_1,...,x_n) = \tau(x_1,...,x_n)$ is appended to the theory. Finally, I planned to develop a natural deduction system for proving theorems in the theory and implement that system in an interactive verifier.

1.2 Previous Work

To my knowledge, R. Boyer and J Moore [Boyer and Moore 1975; Moore 1975] are the only other computer scientists who have pursued a similar line of research. Their objectives, however, were quite different from mine. Their primary goal was to build a completely automatic verifier which could prove as many simple theorems about LISP functions as possible. To accomplish this objective, they defined the semantics of LISP using an approach very similar to my own. First, they created a first-order theory of S-expressions built from the single atom NIL. Then they defined the semantics of a LISP program P containing only total functions by adding the axiom $f(x_1, ..., x_n) = \tau(x_1, ..., x_n)$ for each function definition

$f(x_1, ..., x_n) = \tau(x_1, ..., x_n)$ in P. Their verifier implements a simple set of proof rules derived from these axioms, including rules which perform symbolic evaluation and induction on the structure of the data. A set of heuristics determines which rule is applied at any given point in an attempted proof.

The Boyer-Moore verifier can automatically prove a surprisingly large number of simple theorems, clearly demonstrating the effectiveness of structural induction and symbolic evaluation in program verification. As a special-purpose automatic theorem prover, Boyer and Moore's verifier is an impressive achievement. However, when judged as a PURE LISP verification system, their work suffers from a number of shortcomings, including the following:

1. Their verifier either proves a theorem totally mechanically or fails completely--there is no provision for user guidance. Some very simple LISP theorems cannot be proved using the Boyer-Moore verifier. A typical example of a trivial theorem the Boyer-Moore verifier cannot prove is the following theorem about the standard LISP function **APPEND** [Boyer 1975]:

   ∀x [APPEND(x,APPEND(x,x))=APPEND(APPEND(x,x),x)]
   where
   APPEND(x,y) = IF NULL x THEN y ELSE CONS(CAR(x),APPEND(CDR(x),y)).

2. Their deductive system is not designed to prove arbitrary theorems about arbitrary PURE LISP programs. Their induction rule, for example, is quite weak, being limited to several restricted forms of step-wise induction on S-expressions (binary-trees). Consequently, proofs requiring more general forms of induction (such as the correctness of a merge sorting algorithm presented later in this paper) are beyond the capabilities of their deductive system.

3. To simplify the process of generating proofs, Boyer and Moore limit the data domain of their LISP subset to S-expressions constructed from the single atom NIL. Unfortunately, theorems about LISP functions in this restricted domain are not necessarily true in the more general domain of standard LISP S-expressions. For example, the statement

   ∀x[NILTREE(x)=T]
   where
   NILTREE(x) = NULL(x) OR
   [NILTREE(CAR(x)) AND NILTREE(CDR(x))]

   is a theorem in Boyer and Moore's restricted data domain but obviously is not a theorem in the domain of standard LISP S-expressions (any S-expression containing an atom other than NIL is a counterexample).

4. Since Boyer and Moore's formal system assumes all user-defined functions are total, their verification system only proves partial correctness (i.e. if any function in the program P is not total, any theorem proved about P may not hold). They never defined the semantics of partial functions or developed a method for proving that a particular function is total.

In contrast to Boyer and Moore, my objectives have been to create a consistent formal deductive system capable of proving all theorems of practical interest about PURE LISP programs, and to develop an interactive verifier to help the programmer construct arbitrary proofs within this system. I have not been interested in building any heuristics into the verifier which improve the automatic capabilites of the verifier, but occasionally prevent the programmer from constructing the sequence of proof steps he wants.

1.3 Motivation for Creating TYPED LISP

Early in my research, I discovered that informal, straightforward proofs of simple theorems about LISP functions did not translate directly into formal proofs in my envisioned verification system. In fact, the seemingly trivial task of formally stating many simple theorems turned out to be far more complicated than I anticipated. Consider the ubiquitous sample theorem which asserts that the standard LISP function **REVERSE** has the property that **REVERSE∘REVERSE** is the identity function. The obvious formal statement of this theorem is:

$$\forall x[\mathbf{REVERSE}(\mathbf{REVERSE}(x))=x].$$

Unfortunately, this formulation of the theorem is unsatisfactory, because it is false (any atom other that NIL is a counterexample). **REVERSE** is well-defined only for S-expressions which represent linear lists using the standard encoding. In order to correctly state the theorem, we must define an auxiliary boolean LISP function **LLIST** which is a characteristic function for the subset of S-expressions which represent linear lists. Using **LLIST**, the correct statement of the theorem is:

$$\forall x[\mathbf{LLIST}(x) \supset \mathbf{REVERSE}(\mathbf{REVERSE}(x))=x].$$

The proofs of simple theorems about LISP functions within a first order theory of S-expressions are even more cumbersome. The most concise, natural description of a typical LISP function is not expressed in terms of how it manipulates S-expressions, but in terms of how it operates on some abstract data types which are represented as S-expressions. Unfortunately, the only way to describe LISP functions in a first order theory of S-expressions is in terms of how they affect S-expressions. Proofs which deal with abstract type representations rather than the abstract types themselves have two very serious drawbacks:

1. Many proof steps must be devoted to checking the correctness of code which encodes or decodes the abstract types as concrete representations.
2. Inductive proofs must use induction on the structure of the representations rather than the structure of the abstract types.

As an illustration, consider the following trivial theorem expressing a simple property of the LISP function **APPEND** when applied to linear-lists of atoms (henceforth called atom-lists):

$$\forall x \in \text{atom-lists}\ [\mathbf{APPEND}(x,\mathbf{NIL}) = x]$$

where:

$$\mathbf{APPEND}(x,y) =$$
$$\quad \mathbf{IF\ NULL}(x)\ \mathbf{THEN}\ y\ \mathbf{ELSE\ CONS}(\mathbf{CAR}(x),\mathbf{APPEND}(\mathbf{CDR}(x),y))\ .$$

The proof of this theorem is extremely easy in the theory of atom-lists. We merely apply induction on the structure of x. The base step, x=NIL, is trivial:

$$\mathbf{APPEND}(x,\mathbf{NIL}) = \mathbf{APPEND}(\mathbf{NIL},\mathbf{NIL}) = \mathbf{NIL}$$

by symbolic evaluation. For the induction step, we must show that for any atom-list v, the statement:

$$\forall u \in \text{atoms}\ [\mathbf{APPEND}(\mathbf{CONS}(u,v),\mathbf{NIL}) = \mathbf{CONS}(u,v)]$$

follows from the induction hypothesis:

$$\mathbf{APPEND}(v,\mathbf{NIL}) = v\ .$$

But, symbolic evaluation reduces:

$$\forall u \in \text{atoms}\ [\mathbf{APPEND}(\mathbf{CONS}(u,v),\mathbf{NIL}) = \mathbf{CONS}(u,v)]$$

to:

$$\forall u \in \text{atoms}\ [\mathbf{CONS}(u,\mathbf{APPEND}(v,\mathbf{NIL})) = \mathbf{CONS}(u,v)]$$

which is an immediate consequence of the induction hypothesis and the substitution of equals for equals. Q.E.D.

The proof of the same theorem in the theory of S-expressions is less straightforward. First, in order to correctly state the theorem in terms of S-expressions, we must define a boolean-valued function **ATOMLIST** which is a characteristic function for the set of S-expressions which represent linear-lists of atoms:

ATOMLIST(x) = IF NULL(x) THEN T
ELSE IF ATOM(x) THEN NIL
ELSE ATOM(CAR(x)) AND ATOMLIST(CDR(x))

Using this definition, the theorem can be written:

$$\forall x \in \text{S-expressions}\ [\text{ATOMLIST}(x) \supset \text{APPEND}(x,\text{NIL}) = x]\ .$$

As before, the proof of the theorem proceeds by induction on the structure of x. The base step, x ∈ atoms, splits into two cases: x=NIL and x≠NIL. The first case is identical to the base step of the atom-list proof. In the second case, x≠NIL, symbolic evaluation reduces:

$$\text{ATOMLIST}(x) = T \supset \text{APPEND}(x,\text{NIL}) = x$$

to:

$$\text{NIL} = T \supset \text{APPEND}(x,\text{NIL}) = x$$

which is an immediate consequence of the axiom NIL≠T. For the induction step we must prove that for any S-expressions u and v:

$$\text{ATOMLIST}(\text{CONS}(u,v)) = T \supset \text{APPEND}(\text{CONS}(u,v),\text{NIL}) = \text{CONS}(u,v)$$

is a consequence of the induction hypotheses:

$$\text{ATOMLIST}(u) = T \supset \text{APPEND}(u,\text{NIL}) = u$$

and

$$\text{ATOMLIST}(v) = T \supset \text{APPEND}(v,\text{NIL}) = v\ .$$

Like the base step, the induction step has two cases: u ∈ atoms and u ¬∈ atoms. In the first case, symbolic evaluation reduces:

$$\text{ATOMLIST}(\text{CONS}(u,v)) = T \supset \text{APPEND}(\text{CONS}(u,v),\text{NIL}) = \text{CONS}(u,v)$$

to:

$$\text{ATOMLIST}(v) = T \supset \text{CONS}(u,\text{APPEND}(v,\text{NIL})) = \text{CONS}(u,v)$$

which an immediate consequence of the first induction hypothesis and the substitution of equals for equals. In the remaining case, u ¬∈ atoms, symbolic evaluation reduces:

$$\text{ATOMLIST}(\text{CONS}(u,v)) = T \supset \text{APPEND}(\text{CONS}(u,v),\text{NIL}) = \text{CONS}(u,v)$$

to:

$$\text{NIL} = T \supset \text{CONS}(u,\text{APPEND}(v,\text{NIL})) = \text{CONS}(u,v)$$

which is an immediate consequence of the axiom NIL≠T. Q.E.D.

It is clear that the proof using induction on S-expressions is longer and less transparent than the proof using induction on atom-lists. Since the inductive structure of S-expressions is different from that of atom-lists, the S-expression proof is forced to examine all cases of S-expressions which do not represent atom-lists and prove that they are not atom-list representations.

The auxiliary function **ATOMLIST** serves as a clumsy mechanism for specifying the implicit data type atom-list. If we included atom-list as a distinct, explicit data type in our programming language and expanded our first-order theory to include atom-lists as well as S-expressions, the informal proof using induction on atom-lists could be formalized directly in our first order system. However, since LISP programs typically involve a wide variety of abstract data types, simply adding a few extra data types such as atom-list to LISP will not eliminate the confusion caused by dealing with abstract data type representations rather than the abstract types themselves. In fact, the more complex that an abstract type is, the more confusing that proofs involving its representations are likely to be. Consequently, I decided that the best solution to this problem is to include a comprehensive data type definition facility in LISP and to formally define the semantics of a program P by creating a first-order theory for the particular data types defined in P. The resulting language TYPED LISP is described in the next chapter.

CHAPTER 2

TYPED LISP

2.1 Informal Description of TYPED LISP

TYPED LISP combines a recursive data type definition facility (similar to those proposed by McCarthy [1963] and Hoare [1973]) with a modified subset of PURE LISP. For the sake of semantic simplicity, TYPED LISP does not permit passing functions as parameters or referencing non-local variables (i.e. dynamic scoping). Furthermore, there is no distinction between equivalent and identical data values; there is only one copy of any data value. A TYPED LISP program consists of a set of data type and function definitions. As in PURE LISP, the program is executed by evaluating some expression containing no variables or undefined function identifiers.

2.1.1. Data Type Definitions

The set of primitive data objects in TYPED LISP is the set of all capital identifiers: {A, B, ..., Z, AA, AB, ..., AZ, BA, ..., AAA, ...}. Data types are simply sets constructed from this set of primitive objects using the rules described below. The primitive type atom consists of all primitive data objects except for NIL, ZERO, TRUE, and FALSE. For notational convenience, we let every capital identifier denote the primitive data type consisting of that identifier, e.g. NIL denotes both the data object NIL and the data type {NIL}. The intended meaning of a capital identifier is always clear from its context.

A data type definition in TYPED LISP has the syntax:

type *type-identifier* ▪ *data-type-expression*

where *type-identifier* is a (lower-case) identifier and a *data-type-expression* is either:

1. An *enumeration* listing a finite set of primitive data objects:

$\{C_1, \ldots, C_n\}$,

e.g.

type boolean = {TRUE, FALSE}.

2. A *construction* defining a set of data objects which are constructed from simpler objects. A construction has the syntax

$$c(s_1{:}T_1, \ldots, s_n{:}T_n)$$

where the constructor c is the type-identifier being defined; $s_1, \ldots, s_n$ are (lower-case) identifiers naming the component selector functions; and $T_1, \ldots, T_n$ are the types of the components, e.g.

type pair = pair(atom1: atom, atom2: atom)

which defines the data type pair consisting of ordered pairs of atoms, and creates the constructor function pair: atom × atom → pair for constructing pairs from atoms, and the selector functions atom1, atom2 : pair → atom for selecting components of a pair.

3. A *disjoint-union*

$$T_1 \cup \ldots \cup T_n$$

defining the data type formed by the union of the disjoint data types $T_1, \ldots, T_n$, e.g.

type ext_pair = NIL ∪ pair .

The disjointness of the subtypes $T_1, \ldots, T_n$ can easily be checked at parse-time.

4. A *recursive-union*

$$B_1 \cup \ldots \cup B_m \cup c_1 \cup \ldots \cup c_n$$

of the disjoint data types $B_1, \ldots, B_m$ (called the base types), and the construction types defined by the recursive constructions $c_1, \ldots, c_n$. Each recursive construction must have at least one component type which contains the type defined by the recursive-union. Some sample recursive-union data type definitions are:

type natnum = ZERO ∪ suc(pred: natnum)
type tree = atom ∪ cons(car: tree, cdr: tree)

The members of the type **natnum** defined above are precisely **ZERO**, **suc(ZERO)**, **suc(suc((ZERO))**, **suc(suc(suc(ZERO)))**, ...; and the members of type **tree** are the S-expressions constructed from the base set of objects **atom**.

Besides all primitive data objects and the primitive data type **atom**, there are several other pre-defined data types in every TYPED LISP program. The universal type **any** consists of all data objects defined in the program. All the other pre-defined types can be described in terms of standard TYPED LISP data type definitions as shown below:

type boolean = {TRUE, FALSE}
type natnum = ZERO U suc(pred: natnum)
type minus = minus(abs: suc)
type integer = minus U natnum .

In contrast to PURE LISP, the false boolean value is denoted by the data object **FALSE** rather than **NIL**. Furthermore, boolean functions must return either **TRUE** or **FALSE**.

2.1.2. Function Definitions

TYPED LISP function definitions have the straightforward syntax

function *function-name* $(p_1{:}T_1, \ldots, p_n{:}T_n){:}T = \xi$

where *function-name* is a (lower-case) identifier naming the function being defined; $p_1, \ldots, p_n$ are (lower-case) identifiers serving as parameters; $T_1, \ldots, T_n$ are the types of the corresponding parameters; T is the type of the range of the function; and ξ is a TYPED LISP expression containing no variables other than the parameters.

Every TYPED LISP function is strict, i.e. it is undefined if any of its arguments is undefined or belongs to the wrong type. The only primitive or implicitly defined functions in TYPED LISP, other than the constructor and selector functions corresponding to every construction type, appear below, along with their domain and range specifications:

function	*domain*	*range*	
equals	**any × any**	**boolean**	
⊂	**any × any**	**boolean**	
and	**boolean × boolean**	**boolean**	
or	**boolean × boolean**	**boolean**	
not	**boolean**	**boolean**	
:*T*	**any**	**boolean**	(for every type *T*)

The functions equals, and, or, and not all have the obvious interpretations. Note that, unlike their PURE LISP counterparts, and and or are call-by-value functions (i.e. they always evaluate both of their arguments). The function ⊂ (written as an infix operator) tests whether or not its first argument is a proper substructure of its second argument. Hence, for any data value t, t ⊂ t returns **FALSE**, while ZERO ⊂ suc(ZERO) returns **TRUE**. For every type T, the function $:T$ (written as a postfix operator) is simply the characteristic function for type T. Given any data object x, $x:T$ returns **TRUE** if x is a member of type T and **FALSE** otherwise.

Every construction type definition $c(s_1:T_1, \ldots, s_n:T_n)$ implicitly defines the constructor function c mapping $T_1 \times \ldots \times T_n$ into type c and the selector functions s_i mapping type c into type T_i, i = 1, ..., n. The constructor function c applied to arguments $x_1, \ldots, x_n$ of types $T_1, \ldots, T_n$, respectively, returns the constructed data object $c(x_1, \ldots, x_n)$. Inversely, each selector function s_i applied to the data object $c(x_1, \ldots, x_n)$ returns x_i.

2.1.3. Expressions

Expressions in TYPED LISP are limited to the following forms:

1. An *if-expression* with syntax:

   if ξ_1 then ξ_2 else ξ_3

   where ξ_1, ξ_2, ξ_3 are expressions.

2. A *case-expression* with syntax:

   T case of ξ
   $T_1: \xi_1$
   ...
   $T_n: \xi_n$

   where T is any data type which is an enumeration, disjoint-union, or recursive-union of the types $T_1, \ldots, T_n$; and $\xi, \xi_1, \ldots, \xi_n$ are expressions.

3. A *function-call* of one of the following four forms:

   $f(\xi_1, \ldots, \xi_n)$
   not ξ_1
   ξ_1 *binary-boolean-operator* ξ_2
   ξ_1 *type-operator* T

where *f* is a function-name (including constructor and selector names); $\xi_1, \ldots, \xi_n$ are expressions; *binary-boolean-operator* is either equals, nequals, $\in$, $\neg\in$, and, or or; *type-operator* is either : or $\neg$: ; and *T* is any data type.

4. A bracketed expression

   [ξ]

   where ξ is an expression.

5. A primitive data object (capital identifier).

In TYPED LISP, expressions are evaluated according to a very simple set of rules. As in nearly every language (e.g. ALGOL W) implementing conditional and case expressions, only the index-expression and the selected alternative expression are evaluated. Since all TYPED LISP functions are strict (except for the conditional and case operators), all function arguments are passed by value--including the arguments of boolean binary-operators, not, type-operators, and all constructor and selector functions. In other words, every argument of a function-call is evaluated before the function-call itself is evaluated. Each of the operators equals, $\in$, and $\neg$:*T* (for each type *T*) simply denotes the primitive TYPED LISP function of the same name. Similarly, the operators nequals, $\neg\in$, and $\neg$:*T* (for each type *T*) denote the the functions not ● equals, not ● $\in$, and not ● :*T*, respectively.

Initially, I enforced the following standard parse-time type checking rules in expressions:

1. The declared type of an argument in a function-call must be a subset of the declared type of the corresponding formal parameter.
2. The declared type of the body of a function must be a subset of the declared type of the function's range.
3. The declared type of the index-expression in a case-expression must be a subset of the type declared at the head of the case-expression. Similarly, the type of the index-expression in an if-expression must be a subset of the type boolean.

However, I quickly discarded the idea when it became apparent that such type restrictions forced the programmer to write awkward, inefficient code in many cases. Consider the following sample program defining the commonly used functions assoc and put for manipulating LISP "a-lists".

```
type tree = atom U join(left: tree, right: tree)
type pair = pair(var: atom, val: tree)
type pair_list = NIL U cons(head: pair, tail: pair_list)
```

```
type ext_pair = NIL U pair

function assoc(v: atom, l: pair_list): ext_pair =
   join case of l
      NIL: NIL
      cons: if var(head(l)) equals v then head(l)
            else assoc(v,tail(l))

function put(v: atom, t: tree, l: pair_list): pair_list =
   pair_list case of l
      NIL: cons(pair(v,t),NIL)
      cons: if var(head(l)) equals v then cons(pair(v,t),tail(l))
            else cons(head(l),put(v,t,tail(l)))
```

Now assume we want to use the expression val(assoc(v,l)) in a context where l always contains a pair p with head(p)=v. The declared type of assoc is ext_pair which is not a subset of the domain of the selector function val. Consequently, the expression is syntactically invalid according to standard parse-time type-checking rules, even though its meaning is clear. If we insist on requiring the type of an argument to be a subset of the declared type of the corresponding formal parameter, we must replace

```
val(assoc(v,l))
```

by

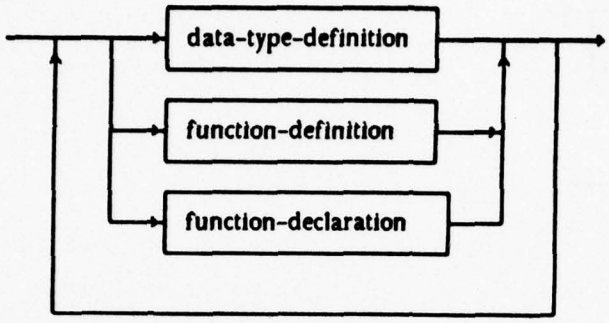
```
ext_pair case assoc(v,l) of
   NIL: some value indicating an error
   pair: val(assoc(v,l))
```

even though the NIL alternative of the case-expression can never be executed. Since situations of the this kind frequently occur in actual programming practice, I relaxed the parse-time type checking rules so that the type of an expression only has to intersect the type required by its context, e.g. the type of an argument must intersect the type of the corresponding formal parameter. Of course, in an actual TYPED LISP implementation, run-time type checking should be done in all those cases where standard parse-time rules are violated. If the user formally proves that a certain run-time error can never occur, then that particular check can be safely eliminated.

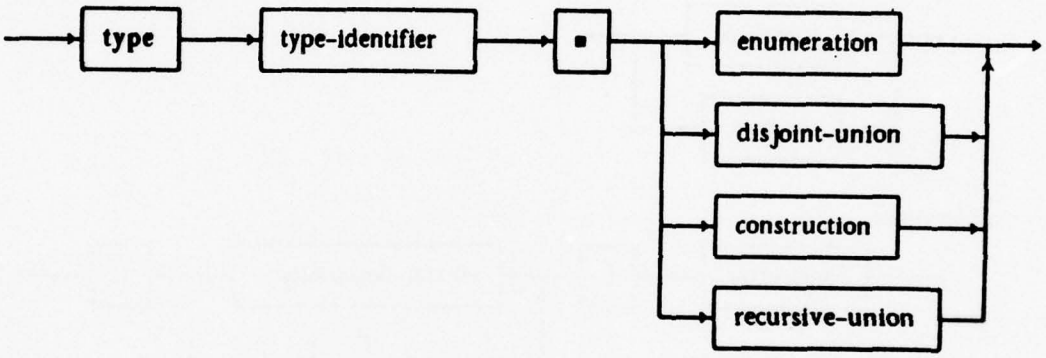
2.2 Syntax of TYPED LISP

The formal syntax description appears below. I have followed Hoare and Wirth's syntax diagram notation as closely as possible using the graphics characters available to me. Non-terminal symbols appear in standard type; terminal symbols appear in **boldface**.

program

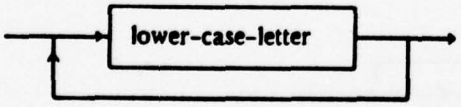


data-type-definition

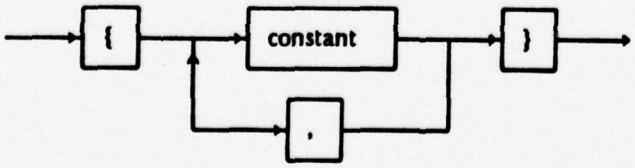


type-identifier

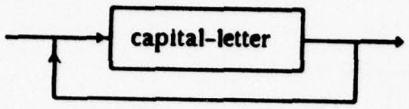
identifier



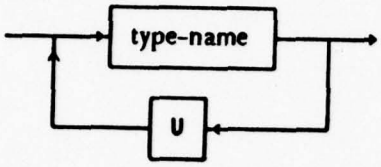
enumeration



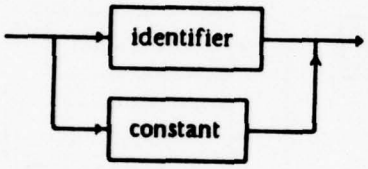
constant



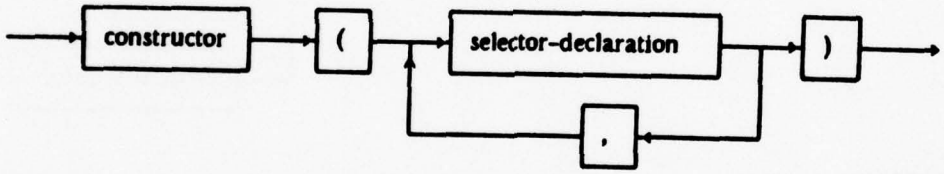
disjoint-union



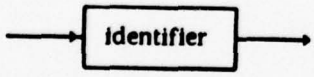
type-name



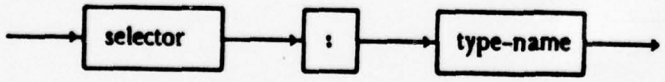
construction



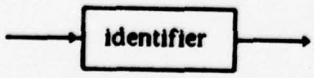
constructor



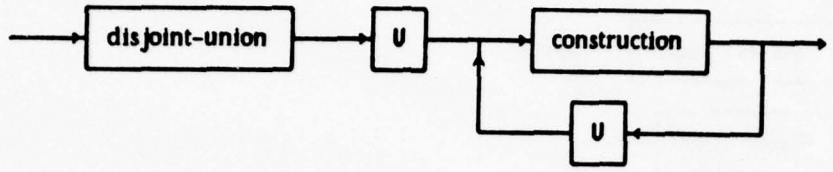
selector-declaration



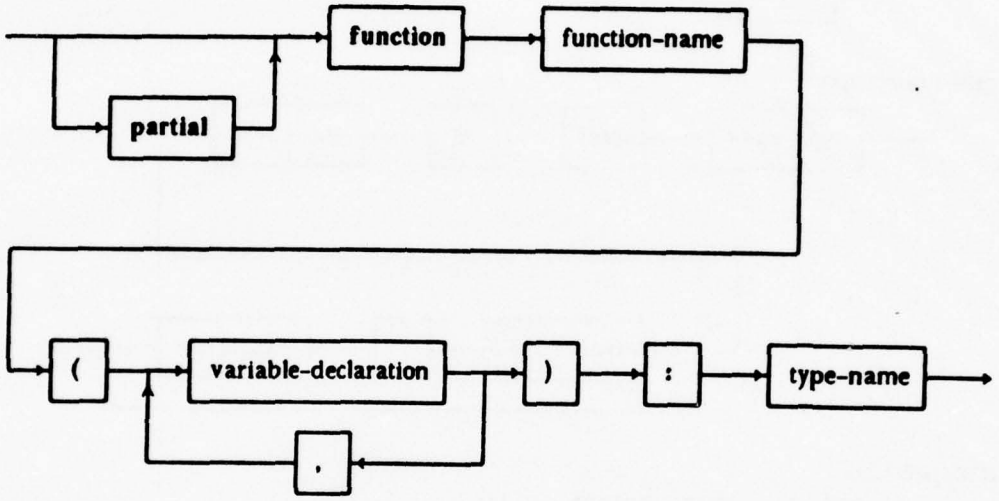
selector



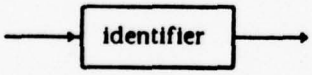
recursive-union



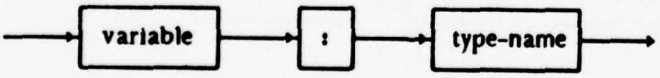
function-declaration



function-name

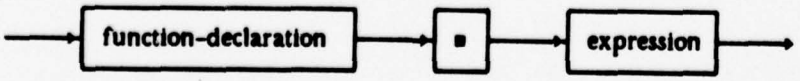


variable-declaration

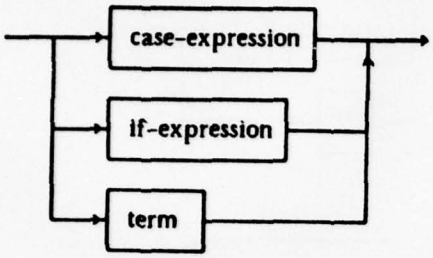


variable

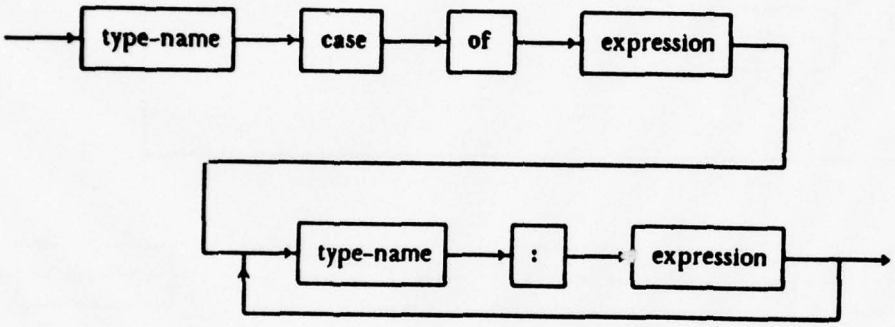
function-definition



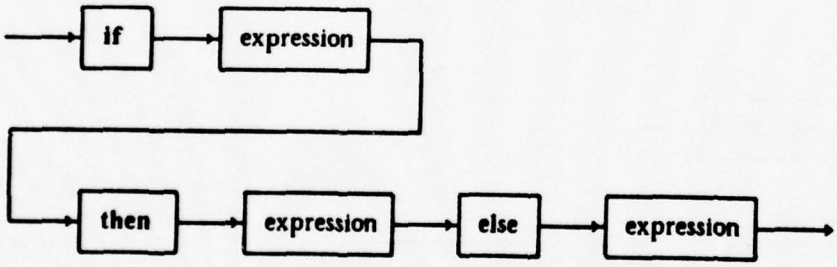
expression



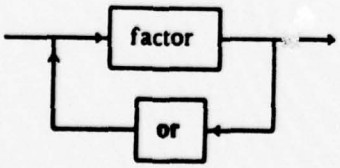
case-expression



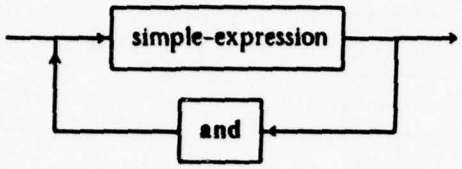
if-expression



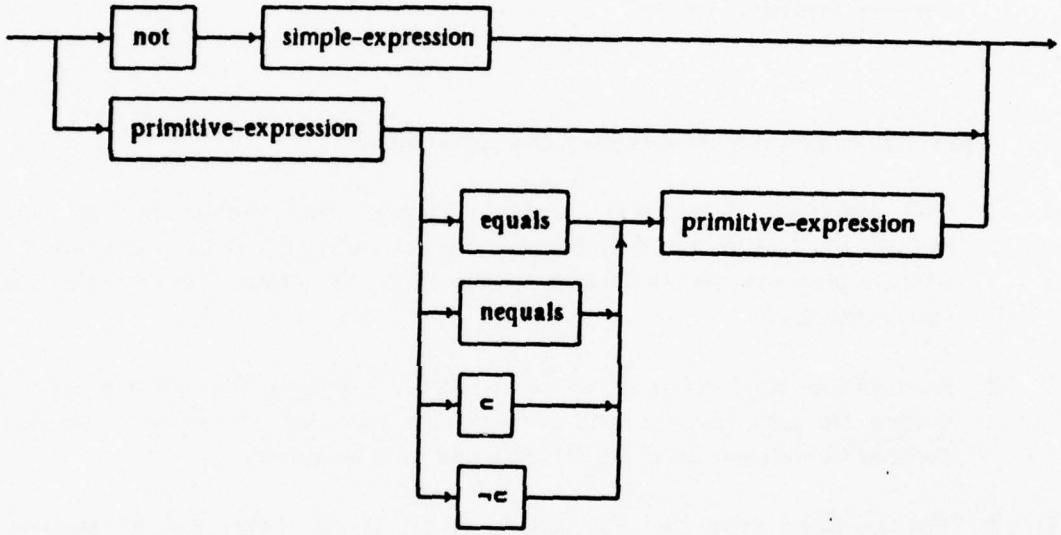
term



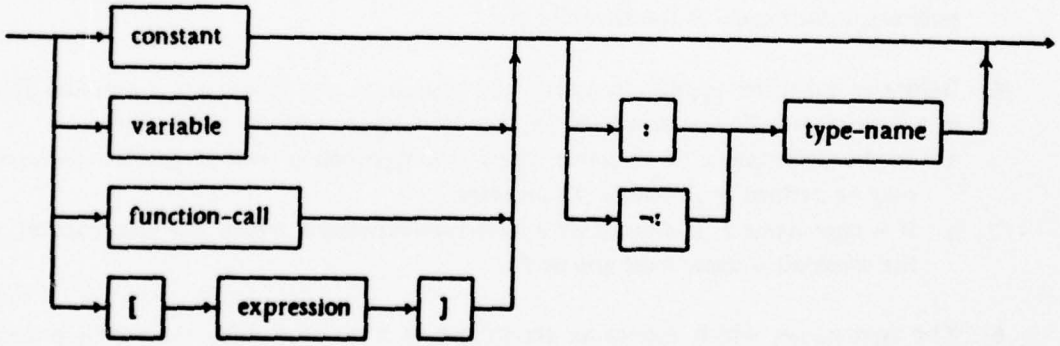
factor



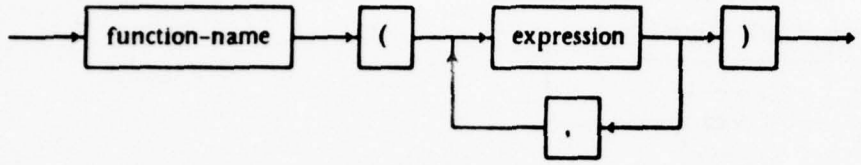
simple-expression



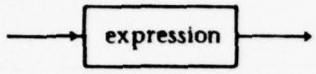
primitive-expression



function-call



execution-expression



The syntax rules which are not context free appear below:

1. The identifiers **atom**, **any**, **boolean**, **natnum**, **suc**, **minus**, **integer**, and all capital-identifiers are pre-defined type-names for every TYPED LISP program P. The selectors **pred** and **abs** are implicitly defined within the constructions of **suc** and **minus** [see Section 2.1.1].

2. An identifier employed as a constructor, selector, type-name, or function name must be unique. No such identifier may be used as a variable. Furthermore, the variables declared within a particular function definition must be distinct.

3. The identifiers **type**, **function**, **declare**, **partial**, **if**, **then**, **else**, **case**, **of**, **and**, **or**, **not**, **equals**, and **nequals** are all reserved; they may not be defined by the user as type-names, selectors, constructors, or function-names.

4. Before an identifier appears in a data-type-expression as a type-name it must be defined in a data-type-definition as a type-name or constructor--with two exceptions:
   a. In the definition of a recursive type T, the type-names used in selector-declarations may be defined anywhere in the program.
   b. If a type-name T is defined by a data-type-expression which is a construction, then the constructor name must also be T.

5. The type-names which appear as alternatives in a disjoint-union must denote disjoint types. We defer the definition of disjoint type-names until the next chapter (Section 2.3.2) where we will continually use that definition in proofs. Despite the fact that the definition for disjointness of type-names is given in the chapter on semantics, it is really a syntactic notion. We can easily check to see whether or not two types are disjoint at parse time.

6. Every declared function must eventually be defined by a function definition.

Furthermore, the parameter lists in must be identical in both cases.

7. A particular variable v may not appear within the expression forming the body of a function-definition unless it is declared in the function-definition.

8. The type-name T heading a case expression ξ must be either be defined as an enumeration, disjoint-union, or a recursive-union. Within the case body of ξ, the case alternatives must be in one-to-one correspondence (including the same ordering) with the subtype alternatives. Each case alternative must begin with the name of the corresponding subtype.

9. A function-name may not appear in a function-call unless it has already been declared as a function-name in a function-declaration, or as a constructor or selector in a construction. In a function-call, the enclosed list of arguments must contain the appropriate number of arguments for the particular function being called.

10. No variables may appear in an execution-expression.

2.3 Semantics of TYPED LISP

Before defining the semantics of TYPED LISP, we must establish some notation for distinguishing between a symbol and what it denotes. In cases where the distinction is necessary, we will underline the symbol when talking about its denotation. On the other hand, when no confusion is possible, we will usually omit the underlining of denotations in the interests of improved readability.

In this section, we will define the meaning of an arbitrary TYPED LISP program P by using the following approach. First, we will define a first-order predicate calculus language L_P (including equality) such that the terms of L_P include all syntactically valid TYPED LISP expressions given the functions and data types defined in P. Then, to define the meaning of terms and formulas in the language L_P, we will construct a standard structure M_P consisting of a data domain and interpretations for all the constant, function, and predicate symbols in the language. We will use M_P to define the meaning of any statement about P written in L_P.

2.3.1. Assertion Language Syntax

Before we can define the syntax of our assertion language, we must review the standard definition for a first order predicate calculus language including equality:

Definition. Given a countably infinite set of variables V, a (possibly empty) set of constant symbols C, a (possibly empty) set of n-ary function symbols F_n for each positive

integer n, and a (possibly empty) set of n-ary predicate symbols for each positive integer n, we define the corresponding first order language L (including equality) as follows. The *terms of* L are defined by the inductive rules:

a. Any variable $v \in V$ is a term.
b. Any constant symbol $c \in C$ is a term.
c. If $t_1, \ldots, t_n$ are terms and $f \in F_n$ (f is an n-ary function symbol), then $f(t_1, \ldots, t_n)$ is a term.

The *formulas of* L are defined by the rules:

a. If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula.
b. If $t_1, \ldots, t_n$ are terms and $p \in P_n$ (p is n-ary predicate symbol), then $p(t_1, \ldots, t_n)$ is a formula.
c. If α and β are formulas, then $(\alpha \wedge \beta)$ is a formula.
d. If α and β are formulas, then $(\alpha \vee \beta)$ is a formula.
e. If α and β are formulas, then $(\alpha \supset \beta)$ is a formula.
f. If α is a formula, then $(\neg\alpha)$ is a formula.
g. If α is a formula and $v \in V$ (v is a variable), then $(\forall v\ \alpha)$ is a formula.
h. If α is a formula and $v \in V$ (v is a variable), then $(\exists v\ \alpha)$ is a formula.

Given a TYPED LISP program P, the assertion language L_P is the first-order language with the following specifications:

1. The variables of L_P are:
   a. All identifiers which are not reserved or defined as type names, constructors, selectors, or function names in P.
   b. Subscripted single letter identifiers, i.e. $a_1, b_1, \ldots, z_1, a_2, \ldots, z_2, \ldots$
2. The constant symbols of L_P are ω, A, B, . . . , Z, AA, AB, . . . , AZ, . . . , AAA, . . . i.e. all valid TYPED LISP constants plus ω.
3. The function symbols of L_P are equals, $\subset$, not, or, and, is-T (for every type name T defined in P), T-case (for each type name T defined in P which is a disjoint or recursive union), and all the function names, selectors, and constructors defined in P. Each function symbol f in L_P takes exactly the same number of arguments as its counterpart in P; hence, not and is-T (for every type T defined in P) take one argument; equals and $\subset$ take two arguments; and T-case takes n+1 arguments where n is the number of subtype alternatives in the disjoint union forming T.
4. There are no predicate symbols in L_P.

We will omit parentheses around formulas whenever convenient. In the absence of parentheses, the precedence of connectives in decreasing order of binding power is $\neg$, $\wedge$, $\vee$, $\supset$. All of the binary connectives ($\wedge$, $\vee$, $\supset$) are right associative.

To make the syntax of L_P and TYPED LISP consistent, we include the following abbreviations in L_P. Let ξ, ψ, α_1, α_2, . . . denote arbitrary terms, and let T denote an arbitrary type. Then:

1. ξ **not** stands for **not**(ξ)
2. ξ **equals** ψ stands for **equals**(ξ,ψ)
3. ξ **nequals** ψ stands for **not(equals**(ξ,ψ))
4. $\xi \subset \psi$ stands for $\subset(\xi,\psi)$
5. $\xi \neg\subset \psi$ stands for **not**($\subset(\xi,\psi)$)
6. $\xi : T$ stands for **is-**T(ξ)
7. $\xi \neg: T$ stands for **not(is-**T(ξ))
8. **if** ξ **then** ψ **else** ζ stands for **boolean-case**(ξ,ψ,ζ)
9. T **case** ξ **of** T_1: α_1 T_2: α_2 . . . T_n: α_n) stands for T**-case**(ξ,α_1,α_2 ...,α_n)
   (where T is defined as the disjoint union of the types $T_1, T_2, \ldots, T_n$)

The operators introduced in the above abbreviations are ranked in decreasing order of precedence in equivalent groups as follows:

{$:T$, $\neg:T$} (for any type T) ≥
{**equals**, **nequals**, $\subset$, $\neg\subset$} ≥
{**not**} ≥
{**or**} ≥
{**and**}

When the syntax of L_P is extended to include the above abbreviations, the set of terms of L_P includes all syntactically valid TYPED LISP expressions given the declarations in P.

For notational convenience, we also introduce the following formula abbreviations where ξ, ψ are arbitrary terms and α, β are arbitrary formulas.

1. $\xi \neq \psi$ stands for $\neg(\xi = \psi)$
2. $\alpha \equiv \beta$ stands for $(\alpha \supset \beta) \wedge (\beta \supset \alpha)$
3. ξ stands for ξ = **TRUE**

The new connective ≡ has lower precedence than the other connectives ($\neg$, $\wedge$, $\vee$, $\supset$). Like the other binary connectives, it is right associative. Of the three abbreviations introduced above, the last one is by far the most important. Abbreviating the formula ξ=**TRUE** by the term ξ allows us to treat boolean expressions as formulas without jeopardizing the soundness of our formal system. In addition, this abbreviation permits us to denote the universally valid formula by the boolean truth value **TRUE** and the the unsatisfiable formula by the boolean truth value **FALSE** without any loss of precision. Adding this abbreviation to L_P does not

make the syntax of L_P ambiguous; a term's context uniquely determines whether or not it abbreviates a formula.

2.3.2. *Assertion Language Semantics*

We will use a standard first order definition of truth for formulas in L_P given interpretations for the constant symbols and function symbols [Enderton 1972]. The formal definitions appear below:

Definition. Given a first order language L, a *structure* M corresponding to L is a quadruple with the following components:

1. A non-empty set |M| called the domain of M.
2. For each constant symbol C in L, a member $\underline{C}$ of |M|.
3. For each n-ary function symbol f in L, an n-ary function $\underline{f}$: $|M|^n \rightarrow |M|$.
4. For each n-ary predicate symbol P, an n-ary relation $\underline{P} \subset |M|^n$.

Definition. Let M be a structure for a first-order language L including equality. An *interpretation function for* M is any function mapping the set of variables in L into |M|.

Definition. Let s be an interpretation function for the structure M. Given M and s, the *meaning* of any term or formula γ in L is $\langle\gamma,M,s\rangle$ where $\langle\ \rangle$ denotes a translation function that, given a structure M and an interpretation function s, maps the formulas in L into truth values (true or false) and the terms of L into elements of |M|. We define the translation function $\langle\ \rangle$ as follows:

1. For any constant C in L,
   $\langle C, M, s\rangle = \underline{C}$
2. For any variable x in L,
   $\langle x, M, s\rangle = s(x)$.
3. For any term in L that is a function call $f(\alpha_1, \ldots, \alpha_n)$,
   $\langle f(\alpha_1, \ldots, \alpha_n), M, s\rangle = \underline{f}(\langle\alpha_1,M,s\rangle, \ldots,\langle\alpha_n,M,s\rangle)$.
4. For any atomic formula in L of the form $\alpha=\beta$,
   $\langle\alpha=\beta, M, s\rangle$ = true if $\langle\alpha,M,s\rangle$ and $\langle\beta,M,s\rangle$ are equal
   = false otherwise.
5. For any atomic formula in L of the form $P(\alpha_1, \ldots, \alpha_n)$,
   $\langle P(\alpha_1, \ldots, \alpha_n), M, s\rangle$ = true if $(\langle\alpha_1,M,s\rangle, \ldots, \langle\alpha_n,M,s\rangle) \in \underline{P}$
   = false otherwise.
6. For any formula in L of the form $\theta \wedge \phi$,
   $\langle\theta \wedge \phi, M, s\rangle$ = true if $\langle\theta,M,s\rangle$ is true and $\langle\phi,M,s\rangle$ is true
   = false otherwise.
7. For any formula in L of the form $\theta \vee \phi$,

$\langle \theta \vee \phi, M, s\rangle$ = true if $\langle \theta, M, s\rangle$ is true or $\langle \phi, M, s\rangle$ is true
= false otherwise.

8. For any formula in L of the form $\theta \supset \phi$,
$\langle \theta \supset \phi, M, s\rangle$ = true if $\langle \theta, M, s\rangle$ is false or $\langle \phi, M, s\rangle$ is true
= false otherwise.
9. For any formula in L of the form $\neg\theta$,
$\langle \neg\theta, M, s\rangle$ = true if $\langle \theta, M, s\rangle$ is false
= false otherwise.
10. For any formula in L of the form $\forall x\theta$,
$\langle \forall x\theta, M, s\rangle$ = true if $\langle \theta, M, s\rangle$ is true for all interpretation functions s' such that $s'(y)=s(y)$ for every variable y distinct from x.
= false otherwise.
11. For any formula in L of the form $\exists x\theta$,
$\langle \exists x\theta, M, s\rangle$ = true if $\langle \theta, M, s\rangle$ is true for some interpretation functions s' such that $s'(y)=s(y)$ for every variable y distinct from x.
= false otherwise.

The translation of a term formula γ corresponds exactly to our intuitive understanding of the meaning of γ given that each free variable v denotes $s(v)$, each constant symbol denotes the corresponding element in M, each function symbol denotes the corresponding function in M, each predicate symbol denotes the corresponding relation in M, and the built-in equality predicate "=" denotes the binary relation $\{(x,x) \mid x \in |M|\}$.

We will use the following terminology concerning structures throughout the sequel.

Definition. We say that a formula α is *true in* M *for* s if and only if $\langle \gamma, M, s\rangle$ = true. We call M a *model for* α (denoted $M \Vdash \alpha$) iff M satifies α for all interpretation functions s. If every formula in a theory (set of formulas) T in L is satified by M, then we say that M *is a model for* T (denoted $M \Vdash T$).

Proofs of metatheorems about out formal system will frequently rely on the following lemma without specifically citing it, since it is intuitively obvious:

Lemma 1.. If two interpretation functions s1 and s2 are identical for all the free variables appearing in a formula or term γ, $\langle \gamma, M, s1\rangle$ is identical to $\langle \gamma, M, s2\rangle$.

Proof. Immediate from the definition of meaning of terms and formulas.

Before constructing the structure corresponding to an arbitrary TYPED LISP program P, we must define the containment and disjointness relations on type names. Intuitively, the elementary or minimal types in a TYPED LISP program P are the primitive data objects (pre-defined types in every program) and the constructor types defined in P. Every other type

defined in P can be uniquely decomposed into a (possibly infinite) union of these elementary types. The disjointness and containment relations on type names can easily be defined in terms of these decompositions. We formalize this approach in the following definitions.

Definition. A type-name which is a capital-identifier or a constructor is *minimal.*

Definition. The *normal form for type name T defined in* P (denoted NF(*T*)) is a set of TYPED LISP minimal type-names defined inductively by the rules:

a. For any type name *C* which is a capital-identifier: NF(*C*) = {*C*}.
b. For any type name *c* which is a constructor defined in P: NF(*c*) = {*c*}.
c. NF(atom) = {*all capital-identifiers except* **NIL, TRUE, FALSE, ZERO**}.
d. NF(any) = {*all constructors*} ∪ {*all capital-identifiers*}.
e. For any type name *T* defined in P as the union (enumeration, disjoint-union, or recursive-union) of the type names $T_1, \ldots, T_n$:

$$NF(T) = NF(T_1) \cup \ldots \cup NF(T_n).$$

Definition. Let *U* and *V* be any type names defined in P. We say that *U* contains *V* (denoted $V \leq U$) iff NF(*V*) is a subset of NF(*U*).

Definition. Two data type-names *U* and *V* are *disjoint* iff NF(*U*) and NF(*V*) are disjoint sets.

The binary relation ≤ satisfies all the defining properties of a reflexive partial ordering except for anti-symmetry ($x \leq y$ and $y \leq x$ implies $x = y$). Two type-names *U* and *V* may be equivalent without being identical (i.e $x \leq y$ and $y \leq x$, but $x \neq y$). In the next section after defining the interpretations for type-names, we will prove that type-name *U* ≤ type-name *V* iff the data type denoted by *U* is a subset of the data type denoted by *V*. Furthermore, we will verify that two type-names are disjoint if and only if the data types they denote are disjoint sets.

Now we are finally ready to define the meaning of an arbitrary TYPED LISP program. For any TYPED LISP program P, we construct the standard structure $\mathbf{M_P}$ as follows:

1. The domain $|\mathbf{M_P}|$ consists of the following symbols:
   a. All underlined capital-identifiers, e. g. $\underline{A}$, $\underline{B}$, . . . , $\underline{Z}$, $\underline{AA}$, . . . Each underlined capital-identifier $\underline{C}$ belongs to every type $\overline{T}$ defined in P such that $C \leq T$ (including *C* of course). Using alternate terminology, *C* belongs to type *T* if and only if $C \in NF(T)$.
   b. The symbol $\underline{\omega}$ which does not belong to any type.
   c. For each construction definition $c(s_1\colon T_1, s_2\colon T_2, \ldots, s_n\colon T_n)$ in P and for any symbols $\underline{\alpha_1}, \ldots, \underline{\alpha_n}$ in $|\mathbf{M_P}|$ of types $T_1, \ldots, T_n$, respectively, the symbol $\underline{c(\alpha_1,\ldots,\alpha_n)}$

is also in $|M_P|$ and belongs to every type T defined in P such that $c \leq T$ (including c of course). In normal form terminology, $\underline{c}(\underline{\alpha_1,\ldots,\alpha_n}) \in |M_P|$ belongs to type T if and only if $c \in NF(T)$.

2. Each constant symbol C in L_P, is assigned the element $\underline{C}$ of $|M_P|$.

3. Each n-ary function symbol f in L_P is assigned an n-ary function $\underline{f} : |M_P|^n \rightarrow |M_P|$ as follows:

   a. We define the functions $\underline{\text{equals}}$, $\underline{\subset}$, $\underline{\text{and}}$, $\underline{\text{or}} : |M_P|^2 \rightarrow |M_P|$ and $\underline{\text{not}} : |M_P| \rightarrow |M_P|$ corresponding to the function symbols equals, $\subset$, and, or, and not, respectively, by:

   $\underline{\text{equals}}(\underline{x},\underline{y}) = \underline{\omega}$ if $\underline{x} = \underline{\omega}$ or $\underline{y} = \underline{\omega}$
   $= \underline{\text{TRUE}}$ if $\underline{x} = \underline{y}$ and $\underline{x},\underline{y} \neq \underline{\omega}$
   $= \underline{\text{FALSE}}$ otherwise

   $\underline{\subset}(\underline{x},\underline{y}) = \underline{\omega}$ if $\underline{x} = \underline{\omega}$ or $\underline{y} = \underline{\omega}$
   $= \underline{\text{TRUE}}$ if $\underline{x}$ textually occurs in $\underline{y}$ and $\underline{x} \neq \underline{y}$
   $= \underline{\text{FALSE}}$ otherwise

   $\underline{\text{and}}(\underline{x},\underline{y}) = \underline{\omega}$ if $\underline{x} \neg\in$ boolean or $\underline{y} \neg\in$ boolean
   $= \underline{\text{TRUE}}$ if $\underline{x} = \underline{\text{TRUE}}$ and $\underline{y} = \underline{\text{TRUE}}$
   $= \underline{\text{FALSE}}$ otherwise

   $\underline{\text{or}}(\underline{x},\underline{y}) = \underline{\omega}$ if $\underline{x} \neg\in$ boolean or $\underline{y} \neg\in$ boolean
   $= \underline{\text{TRUE}}$ if $\underline{x} = \underline{\text{TRUE}}$ or $\underline{y} = \underline{\text{TRUE}}$
   $= \underline{\text{FALSE}}$ otherwise

   $\underline{\text{not}}(\underline{x}) = \underline{\omega}$ if $\underline{x} \neg\in$ boolean
   $= \underline{\text{TRUE}}$ if $\underline{x} = \underline{\text{FALSE}}$
   $= \underline{\text{FALSE}}$ otherwise

   b. For each type T defined in P, we define the function $\underline{\text{is-}T} : |M_P| \rightarrow |M_P|$ interpreting the function symbol is-T by:

   $\underline{\text{is-}T}(\underline{x}) = \underline{\text{TRUE}}$ if $\underline{x} \in$ type T
   $= \underline{\omega}$ if $\underline{x} = \underline{\omega}$
   $= \underline{\text{FALSE}}$ otherwise.

   c. For each type T defined in P as the union (enumeration, disjoint union, or recursive union) of the types $T_1, \ldots, T_n$, we define the function $\underline{T\text{-case}} : |M_P|^{n+1} \rightarrow |M_P|$

interpreting the function symbol T-case by:

$$\underline{T\text{-case}}(\underline{x_0}, \ldots, \underline{x_n}) = \underline{x_i} \text{ if } \underline{x_0} \in T_i,\ 1 \le i \le n,$$
$$= \underline{\omega} \text{ otherwise.}$$

d. For each construction definition $c(s_1: T_1, s_2: T_2, \ldots, s_n: T_n)$ in P, we define the functions $\underline{c} : |M_P|^n \to |M_P|$, and $\underline{s_i} : |M_P| \to |M_P|$ for $i = 1, 2, \ldots, n$ by:

$$\underline{c}(\underline{x_1}, \ldots, \underline{x_n}) = \underline{c(x_1, \ldots, x_n)} \text{ if } \underline{x_i} \in \text{type } T_i,\ 1 \le i \le n,$$
$$= \underline{\omega} \text{ otherwise.}$$

$$\underline{s_i}(\underline{x}) = \underline{\omega} \text{ if } \underline{x} \neg\in \text{type } c$$
$$\underline{s_i}(\underline{c(x_1, \ldots, x_n)}) = \underline{x_i}$$

e. For each function symbol g explicitly defined in a function definition in P,
function $g(x_1 : T_1, \ldots, x_n : T_n) : T = \tau$,

we define the corresponding function $\underline{g} : |M_P|^n \to |M_P|$ by the following process. First, we create an infinite sequence of structures $M_P = \{M_P{}^j \mid j = 0, 1, \ldots\}$ for L_P such that every member $M_P{}^j$ is identical to M_P except for the interpretations assigned to explicitly defined function symbols. Each explicitly defined function name g is interpreted in $M_P{}^j$ by the function $\underline{g^j}$: $|M_P|^n \to |M_P|$ defined by:

$$\underline{g^j}(\underline{x_1}, \ldots, \underline{x_n}) = \langle \tau, M_P{}^{j-1}, s \rangle \text{ if } j > 0 \text{ and } \underline{x_i} \in T_i \text{ for all } i,\ 1 \le i \le n,$$
$$= \underline{\omega} \text{ otherwise,}$$

where and s is any interpretation function mapping x_i into $\underline{x_i}$, $1 \le i \le n$.

From an intuitive viewpoint, $\underline{g^j}$ is simply the function computed by evaluating g to function-call depth j and returning $\underline{\omega}$ if the computation is incomplete. Informally, we want to interpret g as the limit of $\underline{g^j}$ as j approaches ∞. Restated in more precise terms, our goal is to define $\underline{g}$ as the least upper bound of the sequence $G = \{\underline{g^j} \mid j = 0, 1, \ldots\}$ under the usual partial ordering $\sqsubseteq$ on computable functions. To achieve this goal, we must define the partial ordering $\sqsubseteq$ on functions and prove that the least upper bound of the sequence G exists.

Definition. For any two elements $x, y \in |M_P|$, we say x *is less defined or equal to* y

(denoted $x \sqsubseteq y$) iff $x = \omega$ or $x = y$. For any two functions $r,s : |M_P|^n \rightarrow |M_P|$, we say r *is less defined or equal to* s (denoted $r \sqsubseteq s$) iff $r(x_1, \ldots, x_n) \sqsubseteq s(x_1, \ldots, x_n)$ for all $x_1, \ldots, x_n \in |M_P|$. For any two structures M_1, M_2 in the sequence of structures M_P for the program P, we say that M_1 *is less defined or equal to* M_2 (denoted $M_1 \sqsubseteq M_2$) iff the interpretation for every function symbol f in M_1 is less defined or equal to ($\sqsubseteq$) the corresponding interpretation in M_2.

Given these definitions, it is a straightforward task to prove the following lemmas leading to our main theorem.

Lemma 2. Any ascending sequence of elements $X = \{x_i \mid i = 0, 1, ..\}$ in $|M_P|$ has a least upper bound.

Proof. The sequence is either identically ω for all i, or there exists an integer k such that $x_k \neq \omega$ for some $k \geq 0$. In the former case, ω obviously is a least upper bound. In the latter case, $x_i = x_k$ for all $i \geq k$, since no other element in $|M_P|$ is $\geq x_k$. Consequently, x_k is a least upper bound. Q.E.D.

Lemma 3. Any ascending sequence of functions $\{f_i \mid i = 0, 1, ..\}$ in the function space $|M_P|^n \rightarrow |M_P|$ has a least upper bound.

Proof. Let $g : |M_P|^n \rightarrow |M_P|$ be defined by

$$g(x_1, .., x_n) = \text{l.u.b.}\{f_i(x_1, ..., x_n) \mid i = 0, 1, ..\}.$$

We know that $\text{l.u.b.}\{f_i(x_1, ..., x_n) \mid i = 0, 1, ...\}$ exists by the previous lemma, implying that g is well-defined. Furthermore, by the definition of the $\sqsubseteq$ relation on functions, g must be the least upper bound of the sequence of functions $\{f_i \mid i = 0, 1, ...\}$, since for any $x_1, ..., x_n \in |M_P|$, $g(x_1, .., x_n) = \text{l.u.b.}\{f_i(x_1, ..., x_n) \mid i = 0, 1, ..\}$. Q.E.D.

Lemma 4. Let M_1 and M_2 be structures in the sequence $\{M_P{}^j \mid \text{for } j = 0, 1, ...\}$ such that $M_1 \sqsubseteq M_2$. For any term τ in L_P and any interpretation function s for $|M_P|$, $\langle \tau, M_1, s \rangle \sqsubseteq \langle \tau, M_2, s \rangle$.

Proof. By induction on the structure of τ.

Case 1. τ is a constant. This case is trivial since the interpretation of constants is identical in M_1 and M_2.

Case 2. τ is a variable. This case is also trivial since the interpretation of a variable is entirely independent of the structure--it depends only on s.

Case 3. τ is a function call of the form $f(\tau_1, ..., \tau_n)$. By the induction hypothesis $\langle\tau_i, M_1, s\rangle \sqsubseteq \langle\tau_i, M_2, s\rangle$ for i = 1, ..., n.

Subcase 3a. f is case-T for some type name T which is the union of the types T_i, i = 2, ..., n. By the definition of the sequence of structures M_P, we know case-T is interpreted by $\underline{\text{case-}T}$ in every structure in the sequence. Hence, for any structure M in the sequence,

$$\langle\tau, M, s\rangle = \underline{\text{case-}T}(\langle\tau_1, M, s\rangle, ..., \langle\tau_n, M, s\rangle).$$

If $\langle\tau_1, M_1, s\rangle$ does not belong to type T, then $\langle\tau, M_1, s\rangle$ equals $\underline{\omega}$ and the lemma holds. Otherwise, the induction hypothesis implies $\langle\tau_1, M_1, s\rangle = \langle\tau_1, M_2, s\rangle \in T_i$ for some i. As a result,

$$\langle\tau, M_1, s\rangle = \langle\tau_i, M_1, s\rangle \sqsubseteq \langle\tau_i, M_2, s\rangle = \langle\tau, M_2, s\rangle,$$

proving the lemma in this subcase.

Subcase 3b. f is not case-T for any type name T. Consequently, the interpretation for f in every structure in the sequence M_P is strict. Let $\underline{f_1}$ and $\underline{f_2}$ denote the interpretations for f in M_1 and M_2 respectively, and let $T_1 \times \ldots \times T_n$ be the domain for f declared in P. If, for some i, $\langle\tau_i, M_1, s\rangle$ does not belong to T_i, then

$$\langle\tau, M_1, s\rangle = \underline{f_1}(\langle\tau_1, M_1, s\rangle, ..., \langle\tau_n, M_1, s\rangle) = \underline{\omega}$$

and the lemma is obviously true. Otherwise, the induction hypothesis implies that for all i,

$$\langle\tau_i, M_1, s\rangle = \langle\tau_i, M_2, s\rangle \in T_i.$$

Since the interpretation for f in M_1 is less defined or equal to ($\sqsubseteq$) the corresponding interpretation in M_2, we conclude that

$$\begin{aligned}\langle\tau, M_1, s\rangle &= \underline{f_1}(\langle\tau_1, M_1, s\rangle, ..., \langle\tau_n, M_1, s\rangle)\\ &\sqsubseteq \underline{f_2}(\langle\tau_1, M_2, s\rangle, ..., \langle\tau_n, M_2, s\rangle)\\ &= \langle\tau, M_2, s\rangle,\end{aligned}$$

proving the lemma. Q.E.D.

Lemma 5. The sequence $M_P = \{M_P{}^j \mid j = 0, 1, ...\}$ is ascending.

Proof. We must prove that $M_P{}^j \sqsubseteq M_P{}^{j+1}$ for j = 0, 1, ... The proof proceeds by induction on j. Since the implicitly defined function names of P are interpreted identically in all structures in the sequence M_P, we only need to consider the explicitly defined function names of P. Let g be an arbitrary function name

explicitly defined in P. For j = 0, 1, ..., we must show that $g^j \sqsubseteq g^{j+1}$ given that $M_P{}^{j-1} \sqsubseteq M_P{}^j$ when j > 0.

Base case. j = 0. For each explicitly defined function name g, $g^0(x) = \underline{\omega}$ for all x. Hence $g^0 \sqsubseteq g^1$.

Induction step. Given the induction hypothesis that $M_P{}^{j-1} \sqsubseteq M_P{}^j$, we must prove that $g^j \sqsubseteq g^{j+1}$. Let

function $g(x_1 : T_1, \ldots, x_n : T_n) : T = \tau$,

be the function definition in P for g. By the definition of the sequence of structures M_P, the following identity holds for k = 1, 2, ...:

$$g^k(\underline{x_1}, \ldots, \underline{x_n}) = \langle \tau, M_P{}^{k-1}, s \rangle \text{ if } j > 0 \text{ and } \underline{x_i} \in T_i \text{ for all } i,\ 1 \le i \le n,$$
$$= \underline{\omega} \text{ otherwise.}$$

If, for some i, x_i does not belong to type T_i, we get:

$$g^j(\underline{x_1}, \ldots, \underline{x_n}) = \underline{\omega}$$

and the lemma holds. Otherwise,

$$g^j(\underline{x_1}, \ldots, \underline{x_n}) = \langle \tau, M_P{}^{j-1}, s \rangle$$

and

$$g^{j+1}(\underline{x_1}, \ldots, \underline{x_n}) = \langle \tau, M_P{}^j, s \rangle$$

By the induction hypothesis, $M_P{}^{j-1} \sqsubseteq M_P{}^j$. Consequently, the previous lemma (Lemma 5) implies that

$$g^j(\underline{x_1}, \ldots, \underline{x_n}) = \langle \tau, M_P{}^{j-1}, s \rangle \sqsubseteq \langle \tau, M_P{}^j, s \rangle = g^{j+1}(\underline{x_1}, \ldots, \underline{x_n})$$

proving the lemma. Q.E.D.

Theorem 1. For each explicitly defined function symbol g appearing in P, the corresponding sequence of functions G has the following properties:

a. G is an ascending sequence under the partial ordering $\sqsubseteq$.

b. G has a least upper bound.

Proof of a. Immediate from the previous lemma.

Proof of b. Immediate from property a. above and **Lemma 3.**

From the perspective of least fixed-point semantics, the interpretation in M_P for each explicitly defined function symbol g is the least call-by-value fixed point of the recursion equation for g. Given a recursive definition for the function f,

function $f(x_1 : T_1, \ldots, x_n : T_n) : T = \tau$,

a call-by-value fixed-point of the equation is any function $\underline{f}$ which is a standard least fixed-point [see Milner 1973] of the functional τ^* defined by:

$$\underline{\tau^*}[f](x_1, \ldots, x_n) = \text{if } x_1 : T_1 \text{ and } \ldots \text{ and } x_n : T_n \text{ then } \tau \text{ evaluated at } (x_1, \ldots, x_n) \text{ else } \underline{\omega}.$$

See APPENDIX 3 for a discussion of call-by-value least fixed points.

4. Since there are no predicate symbols in L_P, there are no relations in M_P.

Now that we have constructed the structure structure M_P interpreting the function and constant symbols of L_P, we can define the meaning of formulas and terms in L_P in the obvious way. The meaning of a formula or term γ, given some interpretation function s assigning values to the free variables in γ, is simply $\langle \gamma, M_P, s \rangle$. If a formula γ in L_P is true for all interpretation functions s, then γ is a theorem for P.

2.3.3. The Semantics of Program Composition

At this point, it is illuminating to examine how the structure M_P for the program P changes when we add new function or data type definitions to the program. Intuitively, the meaning of a TYPED LISP data type or function definition is dependent only on the data types or functions used in the definition. Consequently, expanding a program by adding new data type or function definitions should not affect the meaning of the data types or functions already defined. Does this property hold for our formal definition of the meaning of TYPED LISP programs? The answer is a qualified yes.

If we create a new program P^* by adding some new function definitions to the original program P, the interpretations for all of the old function symbols are unchanged in the new the structure M_P*. Similarly, if we include new type definitions in P^*, then all of the interpretations for the old function symbols are unchanged when we restrict them to the original domain. However, there are rare cases where a statement in L_P is true in M_P but false in M_P*, and vice-versa. Fortunately, the statements involved are of no practical importance; they are simply statements or consequences of statements asserting the existence of an object of type any not belonging to any other type defined in P. Furthermore, we can

design our formal deductive system so that the provable theorems of L_P are a subset of the provable theorems of L_P* [Statements which are true for M_P but not for M_P* will not be provable from the axioms for M_P.] Consequently, extending a program won't force us to reprove the theorems we have already proved.

CHAPTER 3

A FORMAL DEDUCTION SYSTEM FOR TYPED LISP

3.1 Introduction

Although Section 2.3.2 presents an intuitively plausible definition of the meaning for any statement in the assertion language L_P, we have no systematic way to determine whether a particular statement in L_P is true or false. Unfortunately, Godel's incompleteness theorem implies that there is no effective procedure which will determine whether an arbitrary statement in L_P is true or false in M_P. In fact, Godel's theorem implies the much stronger result that the true statements of M_P are not recursively enumerable. From the standpoint of completeness, the best that we can do is to construct a set of axioms A_P in L_P incorporating all of the fundamental properties of M_P that we understand and use standard first order predicate calculus rules of inference (e.g. Gentzen's Natural Deduction Rules) to prove theorems from the axioms. In practice, the inherent incompleteness of any axiomatization should not be a very serious problem. If a programmer genuinely understands the workings of a program he writes, then he should know at least in principle how to prove that the program is correct from the basic properties of the program data. Furthermore, if our first-order deductive system is constructed with care, virtually every such proof will be formalizable in our deductive system.

3.2 A_P: An Axiom System for the Standard Model

We will now construct a set of axioms A_P for M_P. To demonstrate that A_P is really an axiomatization of M_P, we must informally prove that every axiom in A_P is true in M_P. In the course of these proofs, we will need the following definitions, lemmas, and theorems:

Lemma 6. For any types T_1, T_2: $T_1 \leq T_2$ implies that type T_1 is a subset of type T_n.

Proof. Immediate from the construction of $|M_P|$.

Definition. A *minimal type* T is either a single element type C containing the primitive object $\underline{C}$, or a construction type.

Lemma 7. Any two minimal types with distinct names are disjoint.

Proof. Immediate from the construction of $|M_P|$.

Lemma 8. Every element of $|M_P|$ except $\underline{\omega}$ belongs to a unique minimal type.

Proof. Every $x \in |M_P|$ is either a primitive object or a constructed object. If x is some primitive object $\underline{C}$ distinct from $\underline{\omega}$, then x belongs to the minimal type C; otherwise x is a constructed object and x belongs to the corresponding contruction type. Since all minimal types are disjoint, no element can belong to more than one minimal type. Q.E.D.

Theorem 2. Every type T defined in P is the union of the types denoted by the type names in $NF(T)$.

Proof. Let x be an arbitrary element of T, and let T_x be the minimal type of x. By the construction of $|M_P|$, x belongs to type T if and only if the type name $T_x \in NF(T)$. Q.E.D.

Corollary 2.1. Let x be a data object in $|M_P|$ and let T_x be the minimal type of x. Then x belongs to type T if and only if $T_x \leq T$.

Proof. Immediate.

Corollary 2.2. If the type name T is defined in P as the union (enumeration, disjoint union or recursive union) of type names $T_1, \ldots, T_n$ then for any data object x, x belongs to type T if and only if x belongs to type T_i for some i, $1 \leq i \leq n$.

Proof. By the definition of type name normal forms, $NF(T) = NF(T_1) \cup \ldots \cup NF(T_n)$. Let T_x denote the minimal type of x. By Corollary 2.2, x belongs to type T if and only if the type name $T_x \in NF(T)$. But the latter condition holds if and only if $T_x \in NF(T_i)$ for some i, $1 \leq i \leq n$, which by Corollary 2.2 is equivalent to $x \in T_i$ for some i.

Lemma 9. Let $\{f^j \mid j = 0, 1, \ldots\}$ be an ascending sequence of functions mapping $|M_P|^n$ into $|M_P|$. Let $\{x_1^j \mid j = 0, 1, \ldots\}, \ldots, \{x_n^j \mid j = 0, 1, \ldots\}$ be ascending sequences of elements in $|M_P|$. Then:

l.u.b. $\{\underline{f}^j(x_1^j, ..., x_n^j) \mid j = 0, 1, ...\} =$

l.u.b. $\{\underline{f}^j(\text{l.u.b.}\{x_1^k \mid k = 0, 1, ...\}, ..., \text{l.u.b.}\{x_n^k \mid k = 0, 1, ...\}) \mid j = 0, 1, ...\}$

Proof. Each ascending sequence $\{x_i^j \mid j = 0, 1, ...\}$, $1 \le i \le n$, is either identically ω for all j, or there is some integer k_i such that $x_i^j = z_i \neq \omega$ for all $j \ge k_i$. If the sequence $\{x_i^j \mid j = 0, 1, ...\}$ is identically ω, we set k_i to zero. Let $k = \max\{k_i \mid 1 \le i \le n\}$. Each sequence $\{x_i^j \mid j = 0, 1, ...\}$, $1 \le i \le n$, is constant after the first k elements. Consequently the two sequences:

$\{\underline{f}^j(x_1^j, ..., x_n^j) \mid j = 0, 1, ...\}$

and

$\{\underline{f}^j(\text{l.u.b.}\{x_1^k \mid k = 0, 1, ...\}, ..., \text{l.u.b.}\{x_n^k \mid k = 0, 1, ...\}) \mid j = 0, 1, ...\}$

are identical beyond the first k elements and have identical least upper bounds. Q.E.D.

Definition. The partial ordering $\subset$ on the domain $|M_P|$ is defined by:

$x \subset y$ iff $\underline{\subset}(x,y) = \underline{\text{TRUE}}$

Lemma 10. The the partial ordering $\subset$ on the domain $|M_P|$ is well-founded (*i.e.* no member of the domain has more than a finite number of predecessors).

Proof. Let z be any member of $|M_P|$. By the definition of $|M_P|$, z must be either a primitive object which has no predecessors or a finitely constructed object. But the only predecessors (under the ordering $\subset$) in $|M_P|$ of a finitely constructed object z are simply the objects textually occurring in z. Obviously, there can only be a finite number of such objects (they all must be objects created in the process of constructing z). Q.E.D.

Theorem 3.. For any function definition in P:

function $f(x_1: T_1, \ldots, x_n: T_n): T_0 = \xi$

and any interpretation function s for L_P which maps x_i into some object in T_i, $1 \le i \le n$:

$\langle f(x_1,...,x_n), M_P, s\rangle = \langle \xi(x_1,...,x_n), M_P, s\rangle$

Proof. Let $\underline{x_i}$ denote $s(x_i)$, the interpretation of x_i, and let $\underline{f}^j$ denote the interpretation of f in the structure M_P^j. By the construction of M_P,

$\langle f(x_1, ..., x_n), M_P, s\rangle = \underline{f}(\underline{x_1}, \ldots, \underline{x_n})$

$= \text{l.u.b.}\{\underline{f}^j(\underline{x_1}, \ldots, \underline{x_n}) \mid j = 0, 1, ...\}.$

$$= \text{l.u.b.}\{\langle \xi, M_P{}^j, s\rangle \mid j = 0, 1, \ldots\}.$$

So to prove this axiom is true in M_P, all we have to do is prove the following lemma.

Lemma 11. For any expression τ in L_P and interpretation function s,

$$\langle \tau, M_P, s\rangle = \text{l.u.b}\{\langle \tau, M_P{}^j, s\rangle \mid j = 0, 1, \ldots\}.$$

Proof. By induction on the structure of τ.

Base step: τ is a constant C. This case is trivial since C is interpreted as $\underline{C}$ in M_P and in $M_P{}^j$, $j \geq 0$.

Induction step: τ is a function call of the form $f(\alpha_1, \ldots, \alpha_n)$ where f is a function symbol in L_P and $\alpha_1, \ldots, \alpha_n$ are terms that satisfy the lemma. As before, let f^j denote the interpretation of f in the structure $M_P{}^j$. If f is not explicitly defined in P (defined by a recursion equation), then we define $f^j = f$ for all j. By the l.u.b. lemma (**Lemma 9**):

$$\begin{aligned}\text{l.u.b.}\{\langle \tau, M_P{}^j, s\rangle \mid j = 0, 1, \ldots\} &= \text{l.u.b.}\{\langle f(\alpha_1, \ldots, \alpha_n), M_P{}^j, s\rangle \mid j = 0, 1, \ldots\} \\ &= \text{l.u.b.}\{f^j(\langle \alpha_1, M_P{}^j, s\rangle, \ldots, \langle \alpha_n, M_P{}^j, s\rangle) \mid j = 0, 1, \ldots\} \\ &= \text{l.u.b.}\{f^j(\underline{\alpha_1}, \ldots, \underline{\alpha_n}) \mid j = 0, 1, \ldots\}\end{aligned}$$

where $\underline{\alpha_i}$, $1 \leq i \leq n$, denotes $\text{l.u.b.}\{\langle \alpha_i, M_P{}^k, s\rangle \mid k = 0, 1, \ldots\}$.

Our induction hypothesis states that for $1 \leq i \leq n$, $\text{l.u.b.}\{\langle \alpha_i, M_P{}^j, s\rangle \mid j = 0, 1, \ldots\} = \langle \alpha_i, M_P, s\rangle$.

Hence by the induction hypothesis and the construction of f,

$$\begin{aligned}\text{l.u.b.}\langle \tau, M_P{}^j, s\rangle \mid j = 0, 1, \ldots\} &= \text{l.u.b.}\{f^j(\langle \alpha_1, M_P, s\rangle, \ldots, \langle \alpha_n, M_P, s\rangle) \mid j = 0, 1, \ldots\} \\ &= f(\langle \alpha_1, M_P, s\rangle, \ldots, \langle \alpha_n, M_P, s\rangle) \\ &= \langle f(\alpha_1, \ldots, \alpha_n), M_P, s\rangle \\ &= \langle \tau, M_P, s\rangle\end{aligned}$$

Q.E.D.

A description and justification for each axiom in the theory A_P for the structure M_P follows. Many of the axioms are immediate consequences of the construction of $|M_P|$. In that case, I will simply state "immediate" as the justification.

1. **Primitive data type axioms.**

a. For distinct constant symbols C_1, C_2:
$C_1 \neq C_2$.

Justification: Immediate, since C_1, C_2 are interpreted by the distinct objects $\underline{C_1}$, $\underline{C_2}$, respectively.

b. For every constant symbol C except ω, NIL, TRUE, FALSE, ZERO:
C: atom.

Justification: Immediate, since the only primitive objects in $|M_P|$ not belonging to type atom are $\underline{\text{NIL}}$, $\underline{\text{TRUE}}$, $\underline{\text{FALSE}}$, $\underline{\text{ZERO}}$, and $\underline{\omega}$.

c. For each constructor definition $c(\sigma)$ where c is a constructor and σ is a list of selector-declarations.
$\forall x\ [x: c \supset x\neg:\ \text{atom}]$.

Justification: In $|M_P|$ a constructed object of type c belongs to to type T if and only if $c \leq T$. By the definition of the $\leq$ relation on type names [See Section 2.2], $c \neg\leq$ atom.

d. For every constant symbol C except ω (i.e any capital-identifier):
$\forall x\ [x{=}C \equiv x{:}C]$.

Justification: In $|M_P|$ the only object defined as an element of type C (where C is any capital-indentifier) is $\underline{C}$.

e. $\neg$ NIL: atom
$\neg$ TRUE: atom
$\neg$ FALSE: atom
$\neg$ ZERO: atom

Justification: Immediate.

f. $\forall x\ [x{:}\text{any} \equiv x{\neq}\omega]$.

Justification: Immediate from the definition of $|M_P|$ and is-T.

g. For every type T defined in P:
$\omega{:}\ T \equiv \omega$.

Justification: Immediate from the definition of is-T.

2. **Union axioms.**

For each data type name T_0 defined as a union (disjoint union, enumeration, or recursive union) of the types $T_1, T_2, \ldots, T_n$:

$\forall x\ [x: T_0 \equiv x: T_1 \vee x: T_2 \vee \ldots \vee x: T_n]$.

Justification: Immediate consequence of Corollary 2.2.

3. **Construction axioms.**

a. For each construction definition $c(s_1: T_1, s_2: T_2, \ldots, s_n: T_n)$

1. $\forall x_1, x_2, \ldots, x_n\ [x_1: T_1 \wedge x_2: T_2 \wedge \ldots \wedge x_n: T_n \supset c(x_1, x_2, \ldots, x_n): c]$.

Justification: Immediate from the definition of $|M_P|$.

2. $\forall y\ [y: c \supset c(s_1(y), s_2(y), \ldots, s_n(y)) = y]$.

Justification: From the definition of $|M_P|$, we know that an object y belongs to the constructor type c if and only if $y = c(x_1, \ldots, x_n)$ for some $x_1, \ldots, x_n \in |M_P|$. Furthermore, the definitions of the constructor and selector functions in M_P assert that $c(x_1, \ldots, x_n) = c(x_1, \ldots, x_n)$, and $s_i(c(x_1, \ldots, x_n)) = x_i$, $1 \le i \le n$. Consequently,

$y = c(x_1, \ldots, x_n) = c(x_1, \ldots, x_n) = c(s_1(y), \ldots, s_n(y))$,

proving the desired result.

3. $\forall y\ [y: c \supset s_j(y): T_j]$ (for $j = 1, 2, \ldots, n$).

Justification: Immediate from the construction of $|M_P|$.

4. $\forall x_1, x_2, \ldots, x_n\ [x_1: T_1 \wedge x_2: T_2 \wedge \ldots \wedge x_n: T_n \supset s_j(c(x_1, x_2, \ldots, x_n)) = x_j]$
(for $j = 1, 2, \ldots, n$).

Justification: Immediate from the construction of $|M_P|$.

b. For any distinct constructors c_1, c_2:

$\forall x\ [x: c_1 \supset x\neg: c_2]$

Justification: Since c_1 and c_2 are distinct minimal types, they must be disjoint (by **Lemma 7**).

4. **Induction axiom schema.**

   For any formula $\beta(x)$ with the single free variable x:

   $$\forall y\ [\forall x\ [x \subset y \supset \beta(x)] \supset \beta(y)] \supset \forall z\ (\beta(z))]$$

   where $\beta(y)$ is an arbitrary formula with the single free variable y.

   Justification: This axiom schema simply asserts that the induction principle holds for the domain $|M_P|$ under the partial ordering $\subset$ (at least for statements expressible in L_P). By **Lemma 10**, the partial ordering $\subset$ is well-founded. Consequently, the rule is valid.

5. **Axioms for equals, not, or, and:**

   a. $\forall x\ [x \neq \omega \equiv x \text{ equals } x \equiv \text{TRUE}]$.
   b. $\forall x,y\ [x \neq y \wedge x \neq \omega \wedge y \neq \omega \equiv x \text{ equals } y \equiv \text{FALSE}]$.
   c. $\forall x\ [\omega \text{ equals } x \equiv \omega]$.
   d. $\forall x\ [x \text{ equals } \omega \equiv \omega]$.
   e. not TRUE ≡ FALSE.
   f. not FALSE ≡ TRUE.
   g. not $\omega \equiv \omega$.
   f. $\forall x\ [x\neg\text{: boolean} \supset \text{not}(x) \equiv \omega$.
   h. TRUE and TRUE ≡ TRUE.
   i. TRUE and FALSE ≡ FALSE.
   j. FALSE and TRUE ≡ FALSE.
   k. FALSE and FALSE ≡ FALSE.
   l. $\forall x\ [\omega \text{ and } x \equiv \omega]$.
   m. $\forall x\ [x \text{ and } \omega \equiv \omega]$.
   n. $\forall x,y\ [x\neg\text{: boolean} \vee y\neg\text{: boolean} \supset x \text{ and } y \equiv \omega]$.
   o. TRUE or TRUE ≡ TRUE.
   p. TRUE or FALSE ≡ TRUE.
   q. FALSE or TRUE ≡ TRUE.
   k. FALSE or FALSE ≡ FALSE.
   l. $\forall x\ [\omega \text{ or } x \equiv \omega]$.
   m. $\forall x\ [x \text{ or } \omega \equiv \omega]$.
   n. $\forall x,y\ [x\neg\text{: boolean} \vee y\neg\text{: boolean} \supset x \text{ and } y \equiv \omega]$.

   Justification: Immediate.

6. **Axioms for $\subset$ (containment).**

a. For each construction definition $c(s_1: T_1, s_2: T_2, \ldots, s_n: T_n)$:

1. $\forall y,x_1,x_2,\ldots,x_n\ [y = c(x_1, x_2, \ldots, _n) \wedge y \neq \omega \supset x_j \subset y]$
for $j = 1, 2, \ldots, n$.

2. $\forall y,x_1,x_2,\ldots,x_n\ [(y \neg\subset x_1 \wedge y \neq x_1) \wedge (y \neg\subset x_2 \wedge y \neq x_2) \wedge \ldots \wedge (y \neg\subset x_n \wedge y \neq x_n) \supset y \neg\subset c(x_1, \ldots, x_n)]$.

b. $\forall x\ [x \neq \omega \supset x \neg\subset x]$.

c. $\forall x,y,z\ [x \subset y \wedge y \subset z \supset x \subset z]$.

d. $\forall x\ [\omega \subset x = \omega]$.

e. For any constant C other than ω:
$\forall x\ [x \neq \omega \supset x \neg\subset C]$.

Justification: Immediate from the definition of $\underline{\subset}$.

7. **Axioms for case.**

For each type identfier T defined in P as the disjoint or recursive union (including enumerations) of the data types $T_1, T_2, \ldots$, and T_n:

a. $\forall y,x_1,x_2,\ldots,x_n\ [y: T_i \supset T \text{ case } y \text{ of } T_1: x_1; \ldots, T_i: x_i; \ldots, T_n: x_n = x_i]$
for $i = 1, 2, \ldots, n$.

b. $\forall y,x_1,x_2,\ldots,x_n\ [\neg(y: T) \supset T \text{ case } y \text{ of } T_1: x_1; \ldots, T_i: x_i; \ldots, T_n: x_n = \omega]$

Justification: Immediate from the definition of case-T.

8. **Function definition axioms.**
For each function definition
function $f(x_1: T_1, \ldots, x_n: T_n): T_0 = \xi(x_1, \ldots, x_n)$
where f is a function identifier; $x_1, \ldots, x_n$ are variables; $T_1, T_2, \ldots, T_n$ are types; and $\xi(x_1, \ldots, x_n)$ is an expression containing no variables other than $x_1, \ldots, x_n$.

a. $\forall x_1,x_2,\ldots,x_n\ [\neg(x_1: T_1) \vee \ldots \vee \neg(x_1: T_n) \supset f(x_1, \ldots, x_n) = \omega]$.

Justification. This axiom asserts $\langle f(x_1,\ldots,x_n), M_P, s\rangle$ equals $\underline{\omega}$ if s maps some x_i, $1 \leq i \leq n$, into some object not belonging to type T_i. But this is a trivial consequence of the fact that $\underline{f}$ is undefined ($\underline{\omega}$) if any of its arguments is of the wrong type.

b. $\forall x_1,x_2,\ldots,x_n\ [x_1: T_1 \wedge \ldots \wedge x_1: T_n \supset f(x_1, \ldots, x_n) = \xi(x_1, \ldots, x_n)]$.

Justification: Immediate from **Theorem 3.**

3.3 Completeness of the Axiom System A_P

We have established that M_P is a model for A_P. However, this fact is still no assurance that A_P is a satisfactory axiomatization for M_P. Some important properties of M_P may not be specified by A_P. By the completeness theorem for first-order predicate calculus, we know that a formula α in L_P is provable from A_P if and only if α is true for all models of A_P. Consequently, examining other models of A_P gives us some hints about the completeness of our axiomatization of M_P (by Godel's incompleteness theorem, it cannot be fully complete).

First, let us look at the alternate models for A_P on the standard domain $|M_P|$. Axiom group 8 asserts that all explicitly defined functions in P are call-by-value fixed-points of their defining recursion equations. Nothing in these axioms restricts their interpretations to the least call-by-value fixed-points. As a result, any set of call-by-value fixed-points is a valid interpretation for the explicitly defined function symbols of P.

For example, let P be some TYPED LISP program containing the following function definition:

function loop(x: natnum): natnum = loop(x)

The interpretation for loop in the standard model is the everywhere undefined function. However, any function mapping $|M_P|$ into $|M_P|$ which is undefined for non-integers is a valid interpretation for loop. Consequently, we cannot prove anything about the function loop other than the trivial fact that it is undefined for non-integers.

In general, the axiom set A_P is strong enough to prove the totality of almost any total TYPED LISP function (obviously, we can't escape the fundamental incompleteness inherent in any recursively-enumerable theory dealing with the termination of arbitrary programs), it cannot prove the non-totality of TYPED LISP functions which are undefined for some inputs, or that equivalent non-total TYPED LISP functions are equivalent--except for a few special cases. However, I do not think this weakness of my semantics is a serious disadvantage in practice. Very few functions appearing in everyday programs have domains which are not recursive (interpreters seem to be the most prominent exception). Consequently, if the data type definition facility in a programming language is powerful enough to define any recursive set as a data type, nearly every function encountered in practice can be written as a total function on user-defined data types. (TYPED LISP currently does not have this power, but it could easily be extended so that it did.) Moreover, in order to handle the rare cases where partial functions are of practical significance, it is possible to write recursive definitions for partial functions which have unique call-by-value fixed-points (the trick is to

create recursive functions which return computation sequences rather than single values). Syntactically characterizing important classes of recursive definitions which have unique call-by-value fixed points is an interesting topic for further research.

The other non-standard models of interest are those which have domains which are extensions of $|M_P|$. While the axioms in A_P specify the characteristics of every user-defined type of P in detail, they make no restrictions on the objects belonging only to type any. Consequently, if we extend the program P to P^* by adding some new data type and function definitions, the structure M_P* is a model for A_P if we exclude the interpretations for the new function symbols (function symbols not included in L_P). As a result, any provable theorem for P is true for P^*. Viewed from the perspective of proof theory, the same result is even easier to derive. For any program P^* that is an extension of P, the axiom set A_P is a subset of the axiom set A_P* (axioms corresponding to a particular function or data type definition are generated independently of the definitions context). Hence, any theorem provable from A_P is provable from A_P*. Since M_P* is model for A_P*, we conclude that any provable theorem for P is true for P^*.

CHAPTER 4

A NATURAL DEDUCTION SYSTEM FOR TYPED LISP

4.1 Introduction

Since proving theorems in L_p using a standard first-order deductive system is an exceedingly long and tedious task, I have developed a natural deduction system N_p for proving quantifier-free theorems in L_p from the axiom set A_p. Restricting the programmer to proving quantifier-free theorems does not seem to be a serious limitation. The programmer can convert any statement involving quantifiers to quantifier-free form by using skolem functions, assuming he can write TYPED LISP definitions for the skolem functions introduced.

In the deduction system N_p, proofs normally proceed backwards from the statement to be proved (called a goal) by matching the goal with the conclusion of a rule reducing the proof of the goal to the proof of the rule's premises (called subgoals) which presumably are easier to prove. Although N_p is designed only to prove quantifier-free formulas, we cannot entirely eliminate quantifiers from N_p, since quantifiers are required for the statement of induction hypotheses appearing in N_p proofs. There are no other exceptions to the ban on quantifiers. Henceforth, we will refer to quantifier-free formulas simply as formulas.

For the sake of simplicity, let us adopt the following notation for statements in N_p. All statements in N_p have the form $A \mid \beta$, where β is any formula and $A = \{\alpha_1, \ldots, \alpha_n\}$ is a (possibly empty) set of hypotheses which are formulas or induction hypotheses. $A \mid \beta$ is simply a compact notation for the L_p formula $\alpha_1 \wedge \ldots \wedge \alpha_n \supset \beta$. Induction hypotheses have the form $\forall x_1,\ldots,x_n[A \mid \beta]$ where $A \mid \beta$ is a statement. We will use the customary notation for substitution into expressions and formulas. Given the expression or formula γ, the variable x, and the expression τ; γ^x_τ denotes the result of substituting τ for every free occurrence of x in γ. We will use the analogous notation A^x_τ to denote the result of substituting τ for every free occurrence of x in the hypothesis set A.

We define the formal system N_P for proving theorems $A \mid \beta$ as follows:

1. $\{\} \mid$ TRUE is provable (i.e. is a theorem).

2. Any other goal $A \mid \beta$ is provable in N_P if and only if there is an inference rule in N_P with conclusion $A \mid \beta$ and premises $A_1 \mid \beta_1, \ldots, A_n \mid \beta_n$, where the premises are provable.

The proof system N_P has four classes of inference rules: expression simplification rules, formula simplification rules, goal simplification rules, and general proof rules. While it is possible to derive every rule of N_P from the axiom set A_P and standard first-order predicate calculus deduction, it is a tedious, uninteresting task. I will follow the simpler course of verifying the truth of the rules for the standard model M_P. Since all of the simplification rules (expression, formula, and goal) follow immediately from the definition of M_P and the definition of truth for first-order languages, their justifications are omitted. A description of the rules in each of the four classes follows.

4.2 Expression Simplification Rules

Expression simplification rules have the form $B \mid \xi_1 \Rightarrow \xi_2$ where $B = \{\beta_1, \ldots, \beta_n\}$ is a set of formulas, and ξ_1, ξ_2 are TYPED LISP expressions. The rule $B \mid \xi_1 \Rightarrow \xi_2$ means that any occurrence of the expression ξ_1 in a goal $A \mid \gamma$ may be replaced by ξ_2 provided that B is a subset of A. Formally, the original goal $A \mid \gamma$ is the conclusion of the inference rule and the transformed goal is the premise. The expression rules of N_P appear below:

1. For any expressions ξ_1, ξ_2, ψ:

$\{\psi\} \mid \psi \Rightarrow$ TRUE

$\{\} \mid$ if TRUE then ξ_1 else $\xi_2 \Rightarrow \xi_1$

$\{\} \mid$ if FALSE then ξ_1 else $\xi_2 \Rightarrow \xi_2$

$\{\} \mid$ if ω then ξ_1 else $\xi_2 \Rightarrow \omega$

$\{\psi \neg: \text{boolean}\} \mid$ if ψ then ξ_1 else $\xi_2 \Rightarrow \omega$

2. For every type T that is a type-union of subtypes $T_1, \ldots, T_n$; and any expressions $\xi, \xi_1, \ldots, \xi_n$:

$\{\xi : T_1\}$ | T case ξ of T_1: ξ_1 ... T_n: ξ_n => ξ_i (for i = 1, ..., n)

$\{\xi \neg: T\}$ | T case ξ of T_1: ξ_1 ... T_n: ξ_n => ω (for i = 1, ..., n)

$\{\}$ | T case ω of T_1: ξ_1 ... T_n: ξ_n => ω

3. For any expressions ξ_1, ξ_2, ψ:

$\{\xi_1 = \xi_2\}$ | ξ_1 equals ξ_2 => ξ_1 : any

$\{\xi_1 \neq \xi_2\}$ | ξ_1 equals ξ_2 => not [ξ_1 : any and ξ_2 : any]

$\{\psi$: boolean$\}$ | ψ equals TRUE => ψ

$\{\psi$: boolean$\}$ | TRUE equals ψ => ψ

$\{\psi$: boolean$\}$ | ψ equals FALSE => not [ψ]

$\{\psi$: boolean$\}$ | FALSE equals ψ => not [ψ]

$\{\}$ | $\psi \subset \psi$ => not [ψ : any]

$\{\xi_1 \subset \xi_2\}$ | $\xi_2 \subset \xi_1$ => FALSE

$\{\xi_1 = \xi_2\}$ | $\xi_1 \subset \xi_2$ => not [ξ_1 : any]

$\{\xi_1 \neg\subset \xi_2\}$ | $\xi_1 \subset \xi_2$ => FALSE

4. For any expressions ξ, ψ and any types T_1, T_2 such that no data object of type T_1 ever occurs structurally within an object of type T_2 (a property which is easily determined from the data type definitions):

$\{\xi : T_1, \psi : T_2\}$ | $\xi \subset \psi$ => FALSE .

5. For every construction type definition $C(S_1: T_1, \ldots, S_n: T_n)$ and expressions $\xi_1, \ldots, \xi_n$, and $\psi_1, \ldots, \psi_n$:

$\{\xi_1 : T_1, \ldots, \xi_n : T_n, \psi_1 : T_1 \ldots, \psi_n : T_n\}$ |

$C(\xi_1, \ldots, \xi_n)$ equals $C(\psi_1, \ldots, \psi_n)$ => ξ_1 equals ψ_1 and ... and ξ_n equals ψ_n

$\{C(\xi_1, \ldots, \xi_n) : C\}$ | $\xi_i \subset C(\xi_1, \ldots, \xi_n)$ => TRUE (for i = 1, ..., n)

6. For every rule which rewrites an expression of the form ξ equals ψ, there is a dual rule with the identical premises which rewrites ξ nequals ψ as the opposite boolean value. Similary, for every rule rewriting an expression of the form $\xi \subset \psi$, there is a dual rule rewriting $\xi \neg\subset \psi$.

7. For any primitive object (capital identifier) c, any data type T, and any expressions ξ, ψ:

$\{\}$ | $c : T$ => TRUE (when c is a member of T)

{ } | $c : T$ ⇒ FALSE (when c is not a member of T)
{ } | $\xi : c$ ⇒ ξ equals c
{ξ ¬: T} | $\xi : T$ ⇒ FALSE
{$\xi : T$} | ξ ¬: T ⇒ FALSE
{ξ : any, ψ: any} | [ξ equals ψ] : boolean ⇒ TRUE
{ξ : any, ψ: any} | [ξ nequals ψ] : boolean ⇒ TRUE
{ξ : any, ψ: any} | [$\xi \subset \psi$] : boolean ⇒ TRUE
{ξ : any, ψ: any} | [ξ ¬⊂ ψ] : boolean ⇒ TRUE
{ξ : any} | [$\xi : T$] : boolean ⇒ TRUE
{ξ : any} | [ξ ¬: T] : boolean ⇒ TRUE
{ξ : boolean, ψ: boolean} | [ξ and ψ] : boolean ⇒ TRUE
{ξ : boolean, ψ: boolean} | [ξ or ψ] : boolean ⇒ TRUE
{ξ : boolean} | [not ξ] : boolean ⇒ TRUE

8. For any expression ξ and any types T_1, T_2 where T_1 is a subset of type T_2:

{$\xi : T_1$} | $\xi : T_2$ ⇒ TRUE

9. For any expression ξ and any disjoint types T_1, T_2:

{$\xi : T_1$} | $\xi : T_2$ ⇒ FALSE

10. For every construction type definition $C(S_1 : T_1, \ldots, S_n : T_n)$ and expressions $\xi_1, \ldots, \xi_n$:

{$\xi_1 : T_1, \ldots, \xi_n : T_n$} | $C(\xi_1, \ldots, \xi_n) : C$ ⇒ TRUE

11. For every rule which rewrites $\xi : T$ there is a dual rule with identical premises which rewrites ξ ¬: T as the opposite boolean value.

12. For any expressions ξ, ψ and any type T:

{ξ : boolean} | ξ and TRUE ⇒ ξ
{ξ : boolean} | TRUE and ξ ⇒ ξ
{ξ : boolean} | ξ and FALSE ⇒ FALSE
{ξ : boolean} | FALSE and ξ ⇒ FALSE
{ξ : boolean} | ξ or TRUE ⇒ TRUE
{ξ : boolean} | TRUE or ξ ⇒ TRUE
{ξ : boolean} | ξ or FALSE ⇒ ξ
{ξ : boolean} | FALSE or ξ ⇒ ξ
{ } | not FALSE ⇒ TRUE
{ } | not TRUE ⇒ FALSE
{ } | not [ξ and ψ] ⇒ not [ξ] or not [ψ]
{ } | not [ξ or ψ] ⇒ not [ξ] and not [ψ]

$\{\xi$: **boolean**$\}$ | **not not** $[\xi]$ $\Rightarrow$ ξ
$\{\}$ | **not** $[\xi$ **equals** $\psi]$ $\Rightarrow$ ξ **nequals** ψ
$\{\}$ | **not** $[\xi$ **nequals** $\psi]$ $\Rightarrow$ ξ **equals** ψ
$\{\}$ | **not** $[\xi \subset \psi]$ $\Rightarrow$ $\xi \neg\subset \psi$
$\{\}$ | **not** $[\xi \neg\subset \psi]$ $\Rightarrow$ $\xi \subset \psi$
$\{\}$ | **not** $[\xi : T]$ $\Rightarrow$ $\xi \neg: T$
$\{\}$ | **not** $[\xi \neg: T]$ $\Rightarrow$ $\xi : T$

13. For any variable **x** and any expression ξ which does not contain **x**:
$\{x = \xi\}$ | **x** $\Rightarrow$ ξ

14. For any expression ξ and any data object ζ (i.e. a primitive object or a constructed data object)
$\{\xi = \zeta\}$ | ξ $\Rightarrow$ ζ

15. For every function F(including all selectors, constructors, and the operators **equals**, **nequals**, $\subset$, $\neg\subset$, **and**, **or**, **not**, $:T$ [for any type T], and $\neg:T$ [for any type T]) with domain $T_1 \times \ldots \times T_n$; and any expressions $\xi_1, \ldots, \xi_n$:
$\{\xi_i \neg: T_i\}$ | $F(\xi_1, \ldots, \xi_n)$ $\Rightarrow$ ω (for i = 1, ..., n)
$\{\xi_i = \omega\}$ | $F(\xi_1, \ldots, \xi_n)$ $\Rightarrow$ ω (for i = 1, ..., n)

4.3 Formula Simplification Rules

Formula simplification rules closely resemble expression simplification rules; the only difference is that they rewrite formulas instead of expressions. A formula simplification rule has the syntax $B \mid \alpha_1 \Rightarrow \alpha_2$ where B is a set of hypotheses, and α_1, α_2 are formulas. The rule $B \mid \alpha_1 \Rightarrow \alpha_2$ means that any occurrence of the formula α_1 in a goal $A \mid \gamma$ may be replaced by α_2 if B is a subset of A. Since expressions can abbreviate formulas in our first-order language, expressions denoting formulas may be rewritten by formula simplication rules. A list of the formula simplification rules in N_P follows:

1. For any expressions ξ_1, ξ_2:

$\{\}$ | ω $\Rightarrow$ **FALSE**
$\{\xi_1 \neg:$ **boolean**$\}$ | ξ_1 $\Rightarrow$ **FALSE**
$\{\}$ | $\xi_1 = \xi_1$ $\Rightarrow$ **TRUE**
$\{\xi_2 = \xi_1\}$ | $\xi_1 = \xi_2$ $\Rightarrow$ **TRUE**

$\{\xi_1 \neq \xi_2\} \mid \xi_1 = \xi_2 \Rightarrow$ FALSE

$\{\xi_2 \neq \xi_1\} \mid \xi_1 = \xi_2 \Rightarrow$ FALSE

$\{\xi_1 \subset \xi_2\} \mid \xi_1 = \xi_2 \Rightarrow$ FALSE

$\{\xi_2 \subset \xi_1\} \mid \xi_1 = \xi_2 \Rightarrow$ FALSE

$\{\xi_2 : \text{any}\} \mid \omega = \xi_2 \Rightarrow$ FALSE

$\{\xi_1 : \text{any}\} \mid \xi_1 = \omega \Rightarrow$ FALSE

$\{\} \mid \xi_1 = \text{TRUE} \Rightarrow \xi_1$

$\{\} \mid \xi_1 = \text{FALSE} \Rightarrow \text{not } [\xi_1]$

$\{\} \mid \text{TRUE} = \xi_1 \Rightarrow \xi_1$

$\{\} \mid \text{FALSE} = \xi_1 \Rightarrow \text{not } [\xi_1]$

$\{\} \mid \xi_1 \text{ equals } \xi_2 \Rightarrow \xi_1 = \xi_2 \wedge \xi_1 : \text{any}$

$\{\} \mid \xi_1 \text{ nequals } \xi_2 \Rightarrow \xi_1 \neq \xi_2 \wedge \xi_1 : \text{any} \wedge \xi_2 : \text{any}$

2. For any expressions ξ_1, ξ_2 and any disjoint types T_1 and T_2:

   $\{\xi_1{:}T_1, \xi_2{:}T_2\} \mid \xi_1 = \xi_2 \Rightarrow$ FALSE

3. For every formula rule which rewrites a formula of the form $\xi = \psi$ as α, there is a dual rule with the same hypotheses which rewrites $\xi \neq \psi$ as $\neg\alpha$.

4. For every construction type definition $C(S_1{:}\,T_1, \ldots, S_n{:}\,T_n)$ and expressions $\xi_1, \ldots, \xi_n$, and $\psi_1, \ldots, \psi_n$:

   $\{C(\xi_1, \ldots, \xi_n) : C,\ C(\psi_1, \ldots, \psi_n) : C\} \mid$
   $C(\xi_1, \ldots, \xi_n) = C(\psi_1, \ldots, \psi_n) \Rightarrow \xi_1 = \psi_1 \wedge \ldots \wedge \xi_n = \psi_n$

5. For every formula α, β; and every expression ξ, ψ:

   $\{\alpha\} \mid \alpha \Rightarrow$ TRUE

   $\{\neg\alpha\} \mid \alpha \Rightarrow$ FALSE

   $\{\} \mid \text{TRUE} \wedge \alpha \Rightarrow \alpha$

   $\{\} \mid \alpha \wedge \text{TRUE} \Rightarrow \alpha$

   $\{\} \mid \text{FALSE} \wedge \alpha \Rightarrow$ FALSE

   $\{\} \mid \alpha \wedge \text{FALSE} \Rightarrow$ FALSE

   $\{\} \mid \text{TRUE} \vee \alpha \Rightarrow$ TRUE

   $\{\} \mid \alpha \vee \text{TRUE} \Rightarrow$ TRUE

   $\{\} \mid \text{FALSE} \vee \alpha \Rightarrow \alpha$

   $\{\} \mid \alpha \vee \text{FALSE} \Rightarrow \alpha$

$\{\}$ | FALSE $\supset \alpha$ ≡> TRUE
$\{\}$ | TRUE $\supset \alpha$ ≡> α
$\{\}$ | $\alpha \supset$ TRUE ≡> TRUE
$\{\}$ | $\alpha \equiv$ TRUE ≡> α
$\{\}$ | TRUE $\equiv \alpha$ ≡> α
$\{\}$ | $\alpha \equiv$ FALSE ≡> $\neg\alpha$
$\{\}$ | FALSE $\equiv \alpha$ ≡> $\neg\alpha$
$\{\}$ | $\neg$ TRUE ≡> FALSE
$\{\}$ | $\neg$ FALSE ≡> TRUE
$\{\}$ | $\neg (\xi \equiv \psi)$ ≡> $\xi \not\equiv \psi$
$\{\}$ | $\neg (\xi \not\equiv \psi)$ ≡> $\xi \equiv \psi$
{ξ : boolean} | $\neg \xi$ ≡> not [ξ]
$\{\}$ | $\neg\neg\alpha$ ≡> α
$\{\}$ | $\neg (\alpha \wedge \beta)$ ≡> $\neg\alpha \vee \neg\beta$
$\{\}$ | $\neg (\alpha \vee \beta)$ ≡> $\neg\alpha \wedge \neg\beta$
$\{\}$ | $\neg (\alpha \supset \beta)$ ≡> $\alpha \wedge \neg\beta$

4.4 Goal Simplification Rules

The goal simplification rule denoted

A | α --> B | β,

matches goal A | α and rewrites it as B | β. Formally, B | β is the premise of the rule and A | α is the conclusion. The three goal simplification rules of N_P are:

A $\cup$ {TRUE} | β --> A | β
A $\cup$ {FALSE} | β --> $\{\}$ | TRUE
A $\cup$ $\{\alpha \wedge \gamma\}$ | β --> A $\cup$ $\{\alpha, \gamma\}$ | β

where A is any hypothesis set, and β, α, γ are any formulas.

4.5 General Proof Rules

We will use Gentzen's notation to express many of the general proof rules of N_P. His notation for the inference rule with conclusion | β and premises $A_1 \mid \beta_1, \ldots, A_n \mid \beta_n$ is:

$$\frac{A_1 \mid \beta_1, \ldots, A_n \mid \beta_n}{A \mid \beta}$$

The rules of N_P and their justifications follow:

1. **Equality substitution rule.** The following expression rewrite rules may be applied to any goal with a hypothesis set including the formula $\xi_1 = \xi_2$:

$$\xi_1 \Rightarrow \xi_2$$
$$\xi_2 \Rightarrow \xi_1.$$

Justification. Immediate from the definition of truth for first-order formulas.

Although these rewrite rules resemble expression simplification rules in form, their intended use is different. Simplification rules are designed so they may be applied universally as part of a simplification procedure. Obviously, the rewrite rules above must be applied selectively.

2. **Transitivity of $\subset$ rule.** For any expressions ξ_1, ξ_2, ξ_3; any hypothesis set A; and any formula β:

$$\frac{A \cup \{\xi_1 \subset \xi_2,\ \xi_2 \subset \xi_3,\ \xi_1 \subset \xi_3\} \mid \beta}{A \cup \{\xi_1 \subset \xi_2,\ \xi_2 \subset \xi_3\} \mid \beta}$$

Justification. Immediate from the definition of the function $\subseteq$ in M_P.

3. **Rule of consequence.** For every goal $A \mid \beta$ and formula α:

$$\frac{A \mid \alpha,\ A \cup \{\alpha\} \mid \beta}{A \mid \beta}$$

Justification. Immediate from the definition of truth for first-order formulas.

4. **Hypothesis deletion rule.** For every goal $A \mid \beta$ and formula α:

$$\frac{A \mid \beta}{A \cup \{\alpha\} \mid \beta}$$

Justification. Immediate from the definition of truth for first-order formulas.

5. **Replacement rule.** For any expression ξ, any type T, any formula β, and any hypothesis set A such that no variable in ξ is bound in A (i.e. appears bound in an induction hypothesis):

$$\frac{A \cup \{x:T\} \mid \beta, \quad A \mid \xi:T}{A^x_\xi \mid \beta^x_\xi}.$$

Justification. Immediate from the definition of truth for first-order formulas.

6. **Type split rule.** Let T be the union of types $T_1, \ldots, T_n$. For any hypothesis set A, any formula β, any expression ξ:

$$\frac{A \cup \{\xi : T_1\} \mid \beta, \ \ldots, \ A \cup \{\xi : T_n\} \mid \beta, \quad A \mid \xi : T}{A \mid \beta}$$

Justification. An immediate consequence of Corollary 2.2.

7. **Formula split rule.** For any formula α containing no free variables other than those appearing in the goal $A \mid \beta$:

$$\frac{A \cup \{\alpha\} \mid \beta, \ A \cup \{\neg\alpha\} \mid \beta}{A \mid \beta}$$

Justification. Immediate from the definition of truth for first-order formulas.

8. **Construction rule.** For any formula α, any hypothesis set A, any constructor type c defined by the construction $c(s_1: T_1, \ldots, s_n: T_n)$, and any variables $x_1, \ldots, x_n$ which do not appear in A or α:

$$\frac{A^x_{c(x_1, \ldots, x_n)} \cup \{x_1 : T_1, \ldots, x_n : T_n\} \mid \alpha^x_{c(x_1, \ldots, x_n)}}{A \cup \{x : c\} \mid \alpha}$$

Justification. Immediate from the definition of the constructor type c.

9. **Induction rule.** Let $A \mid \beta$ be any goal with free variables $x_1, \ldots, x_n$ not occurring in any induction hypotheses of A. Let τ be any expression containing no free variables other than those in $A \mid \beta$, and let $z_1, \ldots, z_n$ be variables distinct from all free variables in $A \mid \beta$. Then:

$$\frac{A \cup \{I^*\} \mid \beta}{A \mid \beta}$$

where $\mathbf{I}^* = \forall z_1, \ldots, z_n \, [A^{*x_1,\ldots,x_n}_{z_1,\ldots,z_n} \cup \{\tau^{x_1,\ldots,x_n}_{z_1,\ldots,z_n} \subset \tau\} \mid \beta^{x_1,\ldots,x_n}_{z_1,\ldots,z_n}]$

and A^* = *{all formulas in* A, *but not the induction hypotheses}*.

Justification. This rule formalizes complete structural induction on the value of the expression τ under the well-founded partial ordering $\subset$. To demonstrate that thei rule is valid, we fix the free variables in the goal $A \mid \beta$. Let τ^* denote the corresponding value of the expression τ. Since the partial ordering $\subset$ is well-founded, we may assume the induction hypothesis $\mathbf{I}$ asserting that the goal $A \mid \beta$ is true for all values of the expression $\tau \subset \tau^*$. (In the base case where no such values of the expression τ exist, the induction hypothesis is vacuous.) Let $x_1, \ldots, x_m$ be all the free variables in the goal $A \mid \beta$ such that $x_1, \ldots, x_n$ do not appear free in any induction hypotheses, and let $z_1, \ldots, z_m$ be variables distinct from the variables of $A \mid \beta$. Then the goal augmented by the assumption $\mathbf{I}$ can be formally expressed:

$$A \cup \{\mathbf{I}\} \mid \beta$$

where $\mathbf{I} = \forall z_1, \ldots, z_m \, [A^{x_1,\ldots,x_m}_{z_1,\ldots,z_m} \cup \{\tau^{x_1,\ldots,x_m}_{z_1,\ldots,z_m} \subset \tau\} \mid \beta^{x_1,\ldots,x_m}_{z_1,\ldots,z_m}]$

The induction hypothesis $\mathbf{I}^*$ in the rule above is simply an instantiation of $\mathbf{I}$ which eliminates any nested induction hypotheses in $\mathbf{I}$. If there are no induction hypotheses in A, then $\mathbf{I}$ and $\mathbf{I}^*$ are identical. On the other hand, if A contains induction hypotheses $\eta_1, \ldots, \eta_m$, then the new induction hypothesis $\mathbf{I}$ itself will contain induction hypotheses $\eta'_1, \ldots, \eta'_m$, corresponding to the induction hypotheses $\eta_1, \ldots, \eta_m$ in A. If we instantiate variables $z_{n+1}, \ldots, z_m$ which are free within $\eta'_1, \ldots, \eta'_m$, as $x_{n+1}, \ldots, x_m$, respectively, then the instantiations of the inner induction hypotheses $\eta'_1, \ldots, \eta'_m$ are identical to the induction hypotheses $\eta_1, \ldots, \eta_m$ in A. Consequently, the instantiated hypotheses may be eliminated, making the instantiated $\mathbf{I}$ identical to $\mathbf{I}^*$.

10. Induction instantiation rule. Let $\forall z_1, \ldots, z_n [G \cup \{\nu \subset \mu\} \mid \delta]$ be an induction hypothesis in the goal $A \mid \beta$ where $G = \{\gamma_1, \ldots, \gamma_k\}$; let $\mu_1, \ldots, \mu_n$ be expressions containing no variables other than the free variables of $A \mid \beta$; and let γ denote the formula $\gamma_1 \wedge \ldots \wedge \gamma_k \wedge (\nu \subset \mu)$. Then:

$$\frac{A \mid \gamma^{\nu_1,\ldots,\nu_n}_{\mu_1,\ldots,\mu_n}, \quad A \cup \{\gamma^{\nu_1,\ldots,\nu_n}_{\mu_1,\ldots,\mu_n}\} \mid \beta}{A \mid \beta}$$

Justification. Immediate from the definition of truth for first-order formulas.

11. **Expansion rule.** Let

$f(x_1: T_1, \ldots, x_n: T_n): T \equiv \tau(x_1, \ldots, x_n)$

be a function-definition in P. Then for any goal with a hypothesis set containing the formulas $\xi_1: T_1, \ldots, \xi_n: T_n$, we have the expression rewrite rule:

$$f(\xi_1, \ldots, \xi_n) \Rightarrow \tau(\xi_1, \ldots, \xi_n).$$

Justification. Immediate from Theorem 3.

This proof rule is identical in form to an expression simplification rule. However, in practice it must be applied selectively while simplification rules are applied universally.

12. **Lemma rule.** A provable goal $\{\delta_1, \ldots, \delta_k\} \mid \gamma$ serves as a lemma in following rule. Let δ denote the formula $\delta_1 \wedge \ldots \wedge \delta_k$. For any goal $A \mid \beta$:

$$\frac{A \cup \{\gamma^*\} \mid \beta, \quad A \mid \delta^*, \quad \{\delta_1, \ldots, \delta_k\} \mid \gamma}{A \mid \beta}$$

where γ^* and δ^* are identical to γ and δ, respectively, when the free variables in the latter terms are replaced by a set of expressions containing no variables other than the free variables in $A \mid \beta$.

Justification. Immediate from the definition of truth for first-order formulas.

The most interesting feature of N_P is the power of the induction rule. It allows not only complete induction on the structure of any variable in an assertion, but complete induction on the structure of an arbitrary expression. The strength of this rule permits much simpler correctness proofs of functions with complicated recursive structure such as a function which sorts by successive merging (see Section 5.3).

CHAPTER 5

THE IMPLEMENTED VERIFICATION SYSTEM

5.1 Introduction

While a pure natural deduction system like N_P is a great improvement over standard first-order deduction systems, it is still not a convenient tool for formally proving interesting theorems about TYPED LISP programs. Proof of non-trivial theorems are too long and complicated to be feasible without some mechanical assistance. Consequently, I have developed TLV, an interactive verification system for TYPED LISP (TLV is an acronym for TYPED LISP Verifier), which helps the user construct proofs. My main design goal in creating TLV, was to automate as many of the straightforward steps in a proof as possible--without letting the verifier get trapped in infinite loops or enormous searches in non-trivial cases. Consequently, TLV requires programmer guidance to prove some theorems which some other verifiers such as Boyer and Moore's can prove completely automatically. However, my verifier can prove theorems far beyond the capabilities of completely automatic verifiers with only a modest amount of direction from the programmer. Furthermore, TLV terminates within a reasonable amount of time after every user command.

5.2 Structure of the Verifier

The top-level of the verifier is a command interpreter which accepts instructions from the user. To verify the program the user first instructs the verifier to read the program from a specified file. The verifier responds by parsing the program and constructing a semantic representation (if the program contains no syntax errors) for future use by the verifier. The verifier also generates a set of lemmas (called syntax lemmas) stating each function in the program not declared partial always terminates and returns an object of the proper type. After the verifier has parsed the program, the user types in the theorems he wants to prove and any lemmas he expects the proofs to require. At this point, the user is free to attack the proofs of theorems and lemmas in any order that he wishes. The verifier keeps track of all the dependencies (lemmas used in the course of a proof) of each lemma or theorem that he proves, preventing any circularity. When the user attacks a particular lemma or theorem, the

verification system de-activates any lemmas that depend on the selected goal. Furthermore, the user has the option of de-activating any lemmas that would otherwise be applied automatically.

To prove a theorem, the user specifies the major steps in the proof, one at a time, and the verifier simplifies the new goals generated by each step, reducing many of the new goals to { } | TRUE. All proofs proceed backwards from the goal to be proved by successively replacing each new subgoal by a set of simpler subgoals until no subgoals remain which do not simplify to the form { } | TRUE. Since the verifier is completely interactive, the user has the option of backing up an arbitrary number of steps within his current proof and trying a different sequence of proof steps.

There is a very close correspondence between the proof steps available on the verifier and the general proof rules of N_p. The major difference is that the construction rule and expansion rule are not an available proof steps; they are automatically applied by the simplifier. Minor differences are discussed in the TLV User's Manual, APPENDIX 2.

The heart of the verifier is a goal simplifier which performs the following functions:

1. It reduces the current goal by applying the simplification rules of N_p and an optional collection of rewrite rules (lemmas) provided by the user.
2. It expands function calls when either the expanded expression can be simplified, or the expanded expression itself is a function-call.
3. If the hypothesis $v : T$ appears in the goal, the simplifier applies the construction rule to variable v.

Before the simplifier can apply a simplification rule or a rewrite rule, it must verify that all the type constraints of the rule are satisfied. Consequently, the simplifier includes a type evaluator which takes an expression γ and determines the smallest type containing γ given the goal hypotheses, syntax lemmas, and type rules (lemmas) provided by the user.

User-provided rules can have one of the following forms:

1. $A \cup A^* \mid \xi_1 \Rightarrow \xi_2$ (an expression rewrite rule) where all the the free variables in $A \cup \{\xi_2\}$ occur in $A^* \cup \{\xi_1\}$.
2. $A \cup A^* \mid \beta_1 \Rightarrow \beta_2$ (a formula rewrite rule) where all the the free variables in $A \cup \{\beta_2\}$ occur in $A^* \cup \{\beta_1\}$.
3. $A \cup A^* \mid \xi : T$ (a type rule) where all the free variables in A occur in $A^* \cup \{\xi\}$.

Formally, user-specified rules are simply statements in N_p employed as lemmas. The symbol ">" has no logical significance; it is only syntactic "sugar" making the directionality of the statements as rewrite rules clear.

The simplifier attempts to apply a particular expression or formula rewrite rule by performing the following pattern-matching procedure.

1. It tries to match the left hand side of the rule's conclusion against an expression or formula γ in the current goal. In order for the match to succeed, some substitution instance of the left hand side of the rule must equal γ.
2. If the match in step 1 is successful, the simplifier applies the matching substitution from step 1 to A^* and attempts to match the transformed hypotheses in A^* against hypotheses of the current goal. In order for the match to succed, some substitution instance of the transformed hypothesis set A^* must be a subset of the current goal's hypothesis set.
3. If the match in step 2 is successful, the simplifier replaces the variables in the rule's hypotheses by their bound values from the matching operation in steps 1 and 2 and tries to simplify these hypotheses to **TRUE**, given all the hypotheses of the current goal.
4. If all the rule's hypotheses simplify to **TRUE**, then the rewrite rule is applicable, and the simplifier replaces the variables in the right hand side π of the rule's conclusion by their bound values from the matching operation and substitutes the transformed π for γ in the current goal.

To simplify a goal, the simplifier first simplifies each of the goal's hypotheses and then simplifies the conclusion. If one of the hypotheses simplifies to **FALSE** or the conclusion simplifies to **TRUE**, the verifier replaces the goal by { } | **TRUE**. When simplifying a formula the verifier generally follows a "top-down" simplification strategy, applying the outermost matching rule, since it presumably is the most general applicable transformation. In the interest of efficiency, this strategy is not followed in every case.

The type evaluator computes the type of given expression γ by matching γ against the current goal's hypotheses, type rules, and syntax lemmas. To match the expression against a type rule or syntax lemma, the type evaluator applies essentially the same matching algorithm described above for the simplifier. If more than one type matches γ, the type evaluator returns the intersection of the matched types as the expression's type.

5.3 Demonstration of the Verifier

As a demonstation of how the verifier works, let us trace through a "proof of correctness" for a TYPED LISP program which sorts a linear-list of natural numbers into non-decreasing order by successive merging. The program appears below:

type list = NIL U cons(car: natnum, cdr: list)
[*Comment: type* list *is the set of linear-lists of natural numbers* (natnums)]
type list_of_cons = NIL U join(hd: cons, tl: list_of_cons)
[*Comment: type* list_of_cons *is the set of linear-lists of non-empty* lists]

```
function drop(l: list): list_of_cons ■
    list case l of
        NIL: NIL
        cons: join(cons(car(l),NIL), drop(cdr(l)))
    [Comment: transforms list l into a linear-list of single-element lists]
function lequal(x: natnum, y: natnum): boolean ■ not [ y < x ]
declare function pair_merge(ll: list_of_cons): list_of_cons
    [Comment: declares the function pair_merge so sort1 can call it; the parser cannot handle
    forward references]
function sort1(ll: join): list ■
    list_of_cons case tl(ll) of
        NIL: hd(ll)
        join: sort1(pair_merge(ll))
    [Comment: sorts the natnums contained in the non-empty list_of_cons ll into
    non-decreasing order]
function sort(l: list): list ■
    list case l of
        NIL: NIL
        cons: sort1(drop(l))
    [Comment: sorts the list l into non-decreasing order]
declare function merge(l1: list, l2: list): list
function pair_merge(ll: list_of_cons): list_of_cons ■
    list_of_cons case ll of
        NIL: NIL
        join: list_of_cons case tl(ll) of
            NIL: ll
            join: join(merge(hd(ll),hd(tl(ll))), pair_merge(tl(tl(ll))))
    [Comment: merges successive pairs of lists in the list_of_cons ll]
function merge_cons(l1: cons, l2: cons): cons
function merge(l1: list, l2: list): list ■
    list case l1 of
        NIL: l2
        cons: list case l2 of
            NIL: l1
            cons: merge_cons(l1, l2)
    [Comment: merges lists l1 and l2 into non-decreasing order]
function merge_cons(l1: cons, l2: cons): cons ■
    if lequal(car(l1),car(l2)) then cons(car(l1), merge(cdr(l1),l2))
    else cons(car(l2), merge(l1,cdr(l2)))
    [Comment: merges non-empty lists into non-decreasing order]
```

The TYPED LISP functions used for the purpose of stating and proving that the function

sort is correct are defined below:

```
function length(ll: list_of_cons): natnum ■
    list_of_cons case ll of
        NIL: ZERO
        join: suc(length(tl(ll)))
    [Comment: determines the number of lists in the list_of_cons ll]
function ordered_cons(l: cons): boolean ■
    list case cdr(l) of
        NIL: TRUE
        cons: if lequal(car(l),car(cdr(l))) then ordered_cons(cdr(l))
              else FALSE
    [Comment: determines whether or not the non-empty list l is non-decreasing]
function ordered(l: list): boolean ■
    list case l of
        NIL: TRUE
        cons: ordered_cons(l)
    [Comment: determines whether or not the list l is non-decreasing]
function list_ordered(ll: list_of_cons): boolean ■
    list_of_cons case ll of
        NIL: TRUE
        join: if ordered_cons(hd(ll)) then list_ordered(tl(ll))
              else FALSE
    [Comment: determines whether or not every list in the list_of_cons ll is non-decreasing]
function delete(n: natnum, l: list): list ■
    list case l of
        NIL: NIL
        cons: if n equals car(l) then cdr(l) else cons(car(l), delete(n,cdr(l)))
    [Comment: deletes the natnum n from list l]
function member(n: natnum, l: list): boolean ■
    list case l of
        NIL: FALSE
        cons: if n equals car(l) then TRUE
              else member(n,cdr(l))
    [Comment: determines whether or not natnum n is a member of list l]
function permutation(ll: list, l2: list): boolean ■
    list case ll of
        NIL: l2 equals NIL
        cons: if member(car(ll),l2) then permutation(cdr(ll), delete(car(ll),l2))
              else FALSE
    [Comment: determines whether or not list ll is a permutation of list l2]
function append(ll: list, l2: list): list ■
```

```
    list case l1 of
        NIL: l2
        cons: cons(car(l1), append(cdr(l1),l2))
    [Comment: appends list l2 to the end of list l1]
function list_append(l: list_of_cons): list ■
    list_of_cons case l of
        NIL: NIL
        join: append(hd(l), list_append(tl(l)))
    [Comment: appends together all the lists in l into a single list]
function list_permutation(l1: list_of_cons, l2: list_of_cons): boolean ■
    permutation(list_append(l1), list_append(l2))
    [Comment: determines whether or not the linear-lists of natnums associated with
    list_of_cons l1 and l2 are permutations of each other]
```

The theorems we must prove to establish the correctness of sort are:

```
theorem •1  x: list
            |- ordered(sort(x))

theorem •2  x: list
            |- permutation(sort(x),x);
```

In this section we present the interesting sections of the verifier's proof of theorem •1; the complete correctness proof appears in APPENDIX 1. After the verifier parses the program, it generates the following syntax lemmas asserting that each function in the program is total and returns a value of the proper type:

```
[-1] l: list
    |- drop(l): list_of_cons

[-2] x: natnum,y: natnum
    |- lequal(x,y): boolean

[-3] ll: list_of_cons
    |- pair_merge(ll): list_of_cons

[-4] ll: join
    |- sort1(ll): list

[-5] l: list
    |- sort(l: list): list
```

```
[-6] l1: cons,l2: cons
     |- merge_cons(l1: cons,l2: cons): cons

[-7] l1: list,l2: list
     |- merge(l1: list,l2: list): list

[-8] l1: list_of_cons
     |- length(l1): natnum

[-9] l: cons
     |- ordered_cons(l): boolean

[-10] l: list
      |- ordered(l): boolean

[-11] l1: list_of_cons
      |- list_ordered(l1): boolean

[-12] n: natnum,l: list
      |- delete(n,l): list

[-13] l1: list,l2: list
      |- permutation(l1,l2): boolean

[-14] l1: list,l2: list
      |- append(l1,l2): list

[-15] l: list_of_cons
      |- list_append(l): list

[-16] l1: list_of_cons,l2: list_of_cons
      |- list_permutation(l1,l2): boolean
```

The proofs of all the termination lemmas except for -4 (the termination of the function sort1) are very easy and are omitted from this paper. However, since sort1 uses a non-trivial recursion scheme, the proof of -4 is slightly more complex, requiring the following lemmas (automatically invoked by the simplifier):

```
rule *1  y: join
         *1 suc(ZERO)<length(y)
         |- length(pair_merge(y))<length(y)=>TRUE
```

rule +2 ll: join
⊢ pair_merge(ll): join

We omit the proofs of +1 and +2; both are easy. The proof of -4 relative to +1 and +2 appears below:

User command: **prove -4**
[Comment: this command sets up -4 as the initial goal to be proved]

Verifier response: **PROVING -4**
ll: join
⊢ sort1(ll): list

User command: **induct length(ll)**
[Comment: this command applies induction on the structure of the value of **length(ll)**]

Verifier response: **NEW GOALS:**

[1] ll: join
•1: length(ll): ZERO
&1 [ll': join
length(ll')⊂length(ll)
⊢ sort1(ll'): list]
⊢ sort1(ll): list

SIMPLIFIES TO:

⊢ TRUE

[2] ll: join
•1: length(ll): suc
&1 [ll': join
length(ll')⊂length(ll)
⊢ sort1(ll'): list]
⊢ sort1(ll): list

SIMPLIFIES TO:

ll.hd.car: natnum,ll.hd.cdr: list,ll.tl: list_of_cons
&1 [ll': join
length(ll')⊂length(ll)

```
      ⊢ sort1(ll'): list]
   ⊢ sort1(join(cons(ll.hd.car,ll.hd.cdr),ll.tl)): list
```

1 GOALS REMAIN TO BE PROVED

The formula labeled &1 is the generated induction hypothesis; all the primed variables in an induction hypothesis are implicitly universally quantified over the scope of the hypothesis. The simplifier easily reduces the first goal (the base step of the induction) to ⊢ **TRUE** since **ll: join** implies **length(ll): suc**. In the second goal (the induction step), the simplifier exploits the fact that **ll: join** by creating new variables **ll.hd.car**, **ll.hd.cdr**, and **ll.tl** and replacing every occurrence of **ll** by **join(cons(ll.hd.car,ll.hd.cdr),ll.tl)**.

User command: **type ll.tl**
[Comment: this command performs a case split on the type of the variable **ll.tl**]

Verifier response: **NEW GOALS:**

```
[1] ll.hd.car: natnum, ll.hd.cdr: list, ll.tl: list_of_cons
   *1 ll.tl: NIL
   &1 [ll': join
      length(ll')<length(ll)
      ⊢ sort1(ll'): list]
   ⊢ sort1(join(cons(ll.hd.car,ll.hd.cdr),ll.tl)): list

 SIMPLIFIES TO:

   ⊢ TRUE

[2] ll.hd.car: natnum, ll.hd.cdr: list, ll.tl: list_of_cons
   *1 ll.tl: join
   &1 [ll': join
      length(ll')<length(ll)
      ⊢ sort1(ll'): list]
   ⊢ sort1(join(cons(ll.hd.car,ll.hd.cdr),ll.tl)): list

 SIMPLIFIES TO:

   ⊢ TRUE
```

LEMMA -4 PROVED RELATIVE TO ASSERTIONS: +1 +2

To reduce the second new goal to |- TRUE, the verifier must apply rule +1, rule +2, the induction hypothesis, and termination lemmas -3 and -7 (which the verifier already knows are true).

The proof of the main theorem decomposes into three stages. For the first stage we need the following lemma:

```
rule +3  y: join
         #1 list_ordered(y)
         |- ordered(sort1(y))=>TRUE
```

We will prove +3 as the second stage in the proof of theorem #1. The first stage of the proof proceeds as follows:

User command: **prove #1**

Verifier response:
```
PROVING #1
x: list
|- ordered(sort(x))
```

User command: **type x**
[Comment: this command does a case split on the type of x]

Verifier response:
```
NEW GOALS:
[1] x: list
    #1 x: NIL
    |- ordered(sort(x))

  SIMPLIFIES TO:

    |- TRUE

[2] x: list
    #1 x: cons
    |- ordered(sort(x))

  SIMPLIFIES TO:

    x.cdr: list, x.car: natnum
    |- ordered(sort1(join(cons(x.car,NIL),drop(x.cdr))))

1 GOALS REMAIN TO BE PROVED
```

User command: consequence list_ordered(join(cons(x.car,NIL),drop(x.cdr)))
[Comment: this command applies the rule of consequence using the specified formula]

Verifier response: NEW GOALS:

[1] x.cdr: list, x.car: natnum
#1 list_ordered(join(cons(x.car,NIL),drop(x.cdr)))
⊢ ordered(sort1(join(cons(x.car,NIL),drop(x.cdr))))

SIMPLIFIES TO:

⊢ TRUE

[2] x.cdr: list, x.car: natnum
⊢ list_ordered(join(cons(x.car,NIL),drop(x.cdr)))

SIMPLIFIES TO:

x.cdr: list,x.car: natnum
⊢ list_ordered(drop(x.cdr))

1 GOALS REMAIN TO BE PROVED

User command: induct x.cdr
[Comment: this command applies induction on the structure of x.cdr]

Verifier response: NEW GOALS:

[1] x.cdr: list, x.car: natnum
#1 x.cdr: NIL
&1 [x.cdr': list,x.car': natnum
#1 x.cdr'⊂x.cdr
⊢ list_ordered(drop(x.cdr'))⇒TRUE]
⊢ list_ordered(drop(x.cdr))

SIMPLIFIES TO:

⊢ TRUE

[2] x.cdr: list, x.car: natnum

```
#1 x.cdr: cons
&1 [x.cdr': list,x.car': natnum
  #1 x.cdr'⊂x.cdr
  |- list_ordered(drop(x.cdr'))=>TRUE]
|- list_ordered(drop(x.cdr))
```

SIMPLIFIES TO:

```
|- TRUE
```

PROOF OF #1 RELATIVE TO THE ASSERTIONS: +3

The proof of +3 depends on the following lemmas:

```
rule +4  n.pred: natnum
         |- ZERO⊂suc(n.pred)

rule +5  y: list_of_cons
         #1 list_ordered(y)
         |- list_ordered(pair_merge(y))=>TRUE

rule +6  p: boolean, x: boolean
         |- if p then x else FALSE => p ∧ x
```

: Since the proofs of +4 and +6 are very easy, they are omitted. The proof of +5 will comprise the third and final stage of the proof. The second stage of the proof is:

User command: prove +3

Verifier response:
```
PROVING +3
y: join
#1 list_ordered(y)
|- ordered(sort1(y))
```

User command: induct length(y)
[Comment: this command applies induction on the structure of length(y)]

Verifier response: NEW GOALS:

```
[1] y: join
  #1 length(y): ZERO
  #2 list_ordered(y)
```

```
    &1 [y': join
      #1 length(y')<length(y)
      #2 list_ordered(y')
      |- ordered(sort1(y'))=>TRUE]
    |- ordered(sort1(y))

  SIMPLIFIES TO:

    |- TRUE

[2] y: join
    #1 length(y): suc
    #2 list_ordered(y)
    &1 [y': join
      #1 length(y')<length(y)
      #2 list_ordered(y')
      |- ordered(sort1(y'))=>TRUE]
    |- ordered(sort1(y))

  SIMPLIFIES TO:

    y.tl: list_of_cons, y.hd.cdr: list, y.hd.car: natnum
    #1 ordered_cons(cons(y.hd.car,y.hd.cdr))
    #2 list_ordered(y.tl)
    &1 [y': join
      #1 length(y')<length(join(cons(y.hd.car,y.hd.cdr),y.tl))
      #2 list_ordered(y')
      |- ordered(sort1(y'))=>TRUE]
    |- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.tl)))

1 GOALS REMAIN TO BE PROVED
```

User command: type y.tl
[Comment: this command does a case split on the type of y.tl]

Verifier response:

```
NEW GOALS:
[1] y.tl: list_of_cons, y.hd.cdr: list, y.hd.car: natnum
    #1 y.tl: NIL
    #2 ordered_cons(cons(y.hd.car,y.hd.cdr))
    #3 list_ordered(y.tl)
    &1 [y': join
      #1 length(y')<length(join(cons(y.hd.car,y.hd.cdr),y.tl))
```

```
    #2 list_ordered(y')
    |- ordered(sort1(y'))=>TRUE]
  |- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.tl)))
```

SIMPLIFIES TO:

```
  |- TRUE
```

```
[2] y.tl: list_of_cons, y.hd.cdr: list, y.hd.car: natnum
  #1 y.tl: join
  #2 ordered_cons(cons(y.hd.car,y.hd.cdr))
  #3 list_ordered(y.tl)
  &1 [y': join
    #1 length(y')<length(join(cons(y.hd.car,y.hd.cdr),y.tl))
    #2 list_ordered(y')
    |- ordered(sort1(y'))=>TRUE]
  |- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.tl)))
```

SIMPLIFIES TO:

```
  |- TRUE
```

PROOF OF +3 RELATIVE TO THE ASSERTIONS: +1 +2 +5

The final stage of the proof of theorem •1 presented in this demonstration is the proof of +5. For this portion of the proof, we need the following lemma about **merge_cons**:

```
rule +7  x1: cons, x2: cons
       #1 ordered_cons(x1)
       #2 ordered_cons(x2)
       |- ordered_cons(merge_cons(x1,x2))
```

The proof of +5 follows:

User command: **prove +5**

Verifier response:
```
PROVING +5
y: list_of_cons
#1 list_ordered(y)
|- list_ordered(pair_merge(y))=TRUE
```

User command: **induct y**

[Comment: this command applies induction on the structure of y]

Verifer response: NEW GOALS:

```
[1] y: list_of_cons
    #1 y: NIL
    #2 list_ordered(y)
    &1 [y': list_of_cons
       #1 y'⊂y
       #2 list_ordered(y')
       |- list_ordered(pair_merge(y'))=>TRUE]
    |- list_ordered(pair_merge(y))=TRUE

  SIMPLIFIES TO:

    |- TRUE

[2] y: list_of_cons
    #1 y: join
    #2 list_ordered(y)
    &1 [y': list_of_cons
       #1 y'⊂y
       #2 list_ordered(y')
       |- list_ordered(pair_merge(y'))=>TRUE]
    |- list_ordered(pair_merge(y))

  SIMPLIFIES TO:

    y.tl: list_of_cons, y.hd.cdr: list, y.hd.car: natnum
    #1 ordered_cons(cons(y.hd.car,y.hd.cdr))
    #2 list_ordered(y.tl)
    &1 [y': list_of_cons
       #1 y'⊂join(cons(y.hd.car,y.hd.cdr),y.tl)
       #2 list_ordered(y')
       |- list_ordered(pair_merge(y'))=>TRUE]
    |- list_ordered(
        list_of_cons case of y.tl
          NIL:  join(cons(y.hd.car,y.hd.cdr),y.tl)
          join:  join(merge(cons(y.hd.car,y.hd.cdr),hd(y.tl)),
                   pair_merge(tl(y.tl))))

1 GOALS REMAIN TO BE PROVED
```

User command: type y.tl
[Comment: this command does a case split on the type of y.tl]

Verifier response: NEW GOALS:

```
[1]
   y.tl: list_of_cons, y.hd.cdr: list, y.hd.car: natnum
   #1 y.tl: NIL
   #2 ordered_cons(cons(y.hd.car,y.hd.cdr))
   #3 list_ordered(y.tl)
   &1 [y': list_of_cons
     #1 y'⊂join(cons(y.hd.car,y.hd.cdr),y.tl)
     #2 list_ordered(y')
     ⊢ list_ordered(pair_merge(y'))=>TRUE]
   ⊢ list_ordered(
       list_of_cons case of y.tl
         NIL:  join(cons(y.hd.car,y.hd.cdr),y.tl)
         join:  join(merge(cons(y.hd.car,y.hd.cdr),hd(y.tl)),
                  pair_merge(tl(y.tl))))

  SIMPLIFIES TO:

   ⊢ TRUE

[2] y.tl: list_of_cons, y.hd.cdr: list, y.hd.car: natnum
   #1 y.tl: join
   #2 ordered_cons(cons(y.hd.car,y.hd.cdr))
   #3 list_ordered(y.tl)
   &1 [y': list_of_cons
     #1 y'⊂join(cons(y.hd.car,y.hd.cdr),y.tl)
     #2 list_ordered(y')
     ⊢ list_ordered(pair_merge(y'))=>TRUE]
   ⊢ list_ordered(
       list_of_cons case of y.tl
         NIL:  join(cons(y.hd.car,y.hd.cdr),y.tl)
         join:  join(merge(cons(y.hd.car,y.hd.cdr),hd(y.tl)),
                  pair_merge(tl(y.tl))))

  SIMPLIFIES TO:

   ⊢ TRUE
```

PROOF OF ⌖5 RELATIVE TO THE ASSERTIONS: ⌖7

5.4 Capabilities of the Verification System

The sample theorems proved in the previous example are typical of the theorems which the TLV verifier can prove with a reasonable amount of programmer guidance. Among the other theorems I have proved using the verifier are: the termination of a program implementing a unification algorithm (assuming all variables have been renamed), the equivalence of an iterative algorithm (using a stack) and a simple recursive algorithm for counting the leaves of a binary tree, the total correctness of an extended version (including assignment) of the McCarthy-Painter compiler for arithmetic expressions [McCarthy and Painter 1967], and the total correctness of a very simple set of data base management functions. An extensive set of examples appears in APPENDIX 1.

CHAPTER 6

FURTHER WORK

6.1 Improving the verification system

While the TYPED LISP Verifier is capable of proving many moderately hard theorems with relative little user guidance, it has many deficiencies. In many cases, the current implementation places too large a burden on the user. Proving a theorem about a non-trivial TYPED LISP often requires proving a multitude of trivial lemmas (particularly syntax lemmas) which could be proven completely automatically. In the TYPED LISP verification system, however, the user must manually prove every single trivial lemma (or accept it on faith). Although the proofs involved are very short--usually only one or two steps--it is annoying for the user to have to worry about proving trivial lemmas at all. Consequently, the system would be enhanced by the addition of a fast automatic theorem prover which would attempt to prove--within a designated time limit--all of the syntax lemmas and any other lemmas or theorems designated by the user. While many simple theorems inevitably would stump the automatic prover, many would be proved without requiring any user attention.

Another aspect of the verification system which could be significantly improved is the goal simplifier. Frequently, the simplifier fails to simplify a goal to **TRUE** because it expands a function call too soon, preventing a rule match. Following a strict top-down simplification strategy tends to minimize this problem--but at the prohibitive cost of exceedingly slow execution. Even when following a faster, less effective simplification strategy the simplifier runs very slowly. In the course of simplifying a goal, the simplifier continually repeats simplifications and type computations it has already performed--executing the same laborious pattern matches each time. A hashed representation for formulas and expressions would solve this problem. With hashed representations, the simplifier could build tables storing the simplified form or computed type for an expression after determining it once. Subsequent attempts to simplify an expression or compute its type would find the answer stored in the table. Unfortunately, the language in which the verifier is implemented, UCI LISP, has no convenient hashed representation for formulas or expressions.

6.2 Proving Theorems About Partial Functions

At this point in time, none of the theorems proved on the verifier involve partial functions. As I argued in Section 3.3, partial functions aren't necessary for most practical programming applications. But there are important exceptions. For example, proving the correctness of a compiler for a universal language (in the sense of Church's Thesis), is a significant practical application requiring the use of partial functions. The interpreters defining the semantics of the source and target languages cannot be expressed as total functions on recursive types. In order to prove interesting theorems about partial functions in the deductive system, the partial function definitions must have unique least call-by-value fixed-points. If the definitions are written so the partial functions compute entire computation sequences rather than single values, then they will have a unique least fixed-point. I hope to use this approach to prove the correctness of a compiler for a simple procedural language including arithmetic expressions, assignment, and while-loops.

6.3 Extending TYPED LISP

LISP, in all its current incarnations, is a seriously flawed language for verification purposes. In contrast to PURE LISP and TYPED LISP, the implemented versions of LISP have intractable semantics because they include numerous features designed to make the language more comprehensive and efficient. On the other hand, neither PURE LISP or TYPED LISP is a suitable language for most symbolic computing applications. They are too restrictive. There is a glaring need for a variant of LISP which includes enough practical features for most symbolic computing applications (such as program verification systems), yet excludes constructs with unmanageable semantics. I think TYPED LISP would make an excellent starting point for such a language. As I pointed out in Section 1.3, user-defined data types can significantly simplify the task of writing and verifying programs which manipulate symbolic data. I would like to see an extended version of TYPED LISP implemented, including the following features not present in the current TYPED LISP language:

1. Input/output operations.
2. Global variables.
3. Table functions (equivalent to attaching properties to arbitrary data objects).
4. Data type definitions creating types which are unions of all defined types fitting a particular scheme (permitting, for example, a single append function for data types which have the form of a list).
5. Data type definitions creating types which are arbitrary recursive subsets of defined types.
6. Some means for conveniently treating programs as data.

REFERENCES

Boyer, R. S. (1975) Personal communication.

Boyer, R. S., and J S. Moore (1975) Proving Theorems about LISP Functions. *J. Ass. Comput. Mach. 1*, 129–144.

Enderton (1972) *A Mathematical Introduction to Logic*, Academic Press, New York.

Hoare, C. A. R. (1973) *Recursive Data Structures.* A. I. Memo 223, Computer Science Department, Stanford University.

Kleene, S. C. (1952) *Introduction to Meta-mathematics.* D. Van Nostrand, Inc., Princeton, N. J.

Manna Z. (1974) *Mathematical Theory of Computation*, McGraw-Hill Book Co., New York, 356–416.

McCarthy, J. (1963) A Basis for a Mathematical Theory of Computation, in P. Braffort and D. Hirschberg (eds.), *Computer Programmming and Formal Systems*, North-Holland Publishing Co., Amsterdam, 33–70.

McCarthy, J. and J. A. Painter (1967) Correctness of a Compiler for Arithmetic Expressions. *Proc. Symp. Appl. Math. 19*, 33–41.

Milner, R. (1973) *Models of LCF.* A. I. Memo 186, Computer Science Department, Stanford University.

Moore, J S. (1975) *Computational Logic: Structure Sharing and Proof of Program Properties.* CSL 75-2, Xerox Palo Alto Research Center, Palo Alto, California.

Park, D. (1969) Fixpoint Induction and Proofs of Program Properties, in B. Meltzer and D. Michie (eds.), *Machine Intelligence*, Vol. 5, 59–78, Edinburgh University Press, Edinburgh.

Scott, D. (1970) Outline of a Mathematical Theory of Computation, *4th Annu. Princeton Conf. Inform. Sciences & Syst.*, 169–176.

Scott, D. and C. Strachey (1971) *Towards a Mathematical Semantics for Computer Languages*, Technical Monograph PRC-6, Oxford University, Oxford, England.

APPENDIX 1

SAMPLE PROOFS

A 1.1 Example 1: Iterative REVERSE

```
program revl;

type list = NIL U cons(car: atom, cdr: list)

function app(x: list, y: list): list =
  list case x of
    NIL: y
    cons: cons(car(x),app(cdr(x),y))

function revl(x: list, y: list): list =
  list case x of
    NIL: y
    cons: revl(cdr(x),cons(car(x),y))

function rev(x: list): list = revl(x,NIL)
=
THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:

[-1]
x:list,y:list
|- app(x,y) : list;

[-2]
x:list,y:list
|- revl(x,y) : list;

[-3]
x:list
|- rev(x) : list;


theorem e1
x:list
|- rev(rev(x))=x;

THEOREM 1 ACCEPTED


rule +1
x:list,y:list
|- revl(revl(x,y),NIL)=>revl(y,x);

 +1 ACCEPTED
```

```
prove e1!

PROVING e1
x:list
|- rev(rev(x))=x

CURRENT GOAL:

[1]
x:list
|- rev(rev(x))=x

     SIMPLIFIES TO:

TRUE
PROOF OF e1 RELATIVE TO THE ASSERTIONS:
 +1
UNPROVED RULES:  +1
UNPROVED SYNTAX LEMMAS:  -3 -2 -1

qed;


prove +1;

PROVING +1
x:list,y:list
|- revl(revl(x,y),NIL)=>revl(y,x)


induct x +!


NEW GOALS:

[1]
x:list,y:list
#1 x : NIL
&1 [x':list,y':list
   #1 x' c x
   |- revl(revl(x',y'),NIL)=>revl(y',x')]
|- revl(revl(x,y),NIL)=revl(y,x)

     SIMPLIFIES TO:

TRUE

[2]
x:list,y:list
#1 x : cons
&1 [x':list,y':list
   #1 x' c x
   |- revl(revl(x',y'),NIL)=>revl(y',x')]
|- revl(revl(x,y),NIL)=revl(y,x)

     SIMPLIFIES TO:

TRUE
PROOF OF +1
UNPROVED SYNTAX LEMMAS:  -3 -2 -1

qed;


prove -1;

PROVING -1
```

```
x:list,y:list
|- app(x,y) : list


induct x!


NEW GOALS:

[1]
x:list,y:list
#1 x : NIL
&1 [x':list,y':list
   #1 x' c x
   |- app(x',y') : list]
|- app(x,y) : list

     SIMPLIFIES TO:

TRUE

[2]
x:list,y:list
#1 x : cons
&1 [x':list,y':list
   #1 x' c x
   |- app(x',y') : list]
|- app(x,y) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -1
UNPROVED SYNTAX LEMMAS:  -3 -2

qed;


prove -2;

PROVING -2
x:list,y:list
|- revi(x,y) : list


induct x!


NEW GOALS:

[1]
x:list,y:list
#1 x : NIL
&1 [x':list,y':list
   #1 x' c x
   |- revi(x',y') : list]
|- revi(x,y) : list

     SIMPLIFIES TO:

TRUE

[2]
x:list,y:list
#1 x : cons
&1 [x':list,y':list
   #1 x' c x
```

```
   |- revl(x',y') : list]
|- revl(x,y) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -2
UNPROVED SYNTAX LEMMAS:  -3

qed;


prove -3!

PROVING -3
x:list
|- rev(x) : list

CURRENT GOAL:

[1]
x:list
|- rev(x) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -3

qed;
```

A 1.2 Example 2: Total correctness of FLATTEN

```
program nflat;

type tree = atom U join(left: tree,  right: tree)
type list = NIL U cons(car: atom,  cdr: list)

function fast_flat1(t: tree, l: list): list =
    tree case t of
        atom:  cons(t,l)
        join:  fast_flat1(left(t),fast_flat1(right(t),l))
function fast_flat(t: tree): list =  fast_flat1(t,NIL)
function append(l1: list, l2: list): list =
    list case l1 of
        NIL: l2
        cons: cons(car(l1),append(cdr(l1),l2))
function slow_flat(t: tree): list =
    tree case t of
        atom: cons(t,NIL)
        join: append(slow_flat(left(t)),slow_flat(right(t)))
=
THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:

[-1]
t:tree,l:list
|- fast_flat1(t,l) : list;

[-2]
t:tree
|- fast_flat(t) : list;

[-3]
l1:list,l2:list
|- append(l1,l2) : list;
```

```
[-4]
t:tree
|- slow_flat(t) : list;


prove -1;

PROVING -1
t:tree,l:list
|- fast_flat1(t,l) : list


induct t!


NEW GOALS:

[1]
t:tree,l:list
#1 t : atom
&1 [t':tree,l':list
   #1 t' < t
   |- fast_flat1(t',l') : list]
|- fast_flat1(t,l) : list;

     SIMPLIFIES TO:

TRUE

[2]
t:tree,l:list
#1 t : join
&1 [t':tree,l':list
   #1 t' < t
   |- fast_flat1(t',l') : list]
|- fast_flat1(t,l) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -1
UNPROVED SYNTAX LEMMAS:  -4 -3 -2

qed;


prove -2;

PROVING -2
t:tree
|- fast_flat(t) : list


!


NEW GOALS:

[1]
t:tree
|- fast_flat(t) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -2
UNPROVED SYNTAX LEMMAS:  -4 -3
```

```
qed;


prove -3;

PROVING -3
l1:list,l2:list
|- append(l1,l2) : list


induct l1!


NEW GOALS:

[1]
l1:list,l2:list
#1 l1 : NIL
&1 [l1':list,l2':list
   #1 l1' c l1
   |- append(l1',l2') : list]
|- append(l1,l2) : list

     SIMPLIFIES TO:

TRUE

[2]
l1:list,l2:list
#1 l1 : cons
&1 [l1':list,l2':list
   #1 l1' c l1
   |- append(l1',l2') : list]
|- append(l1,l2) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -3
UNPROVED SYNTAX LEMMAS:  -4

qed;


prove -4;

PROVING -4
t:tree
|- slow_flat(t) : list
  .
induct t!


NEW GOALS:

[1]
t:tree
#1 t : atom
&1 [t':tree
   #1 t' c t
   |- slow_flat(t') : list]
|- slow_flat(t) : list

     SIMPLIFIES TO:
```

```
TRUE

[2]
t:tree
#1 t : join
&1 [t':tree
   #1 t' c t
   |- slow_flat(t') : list]
|- slow_flat(t) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -4

qed;


theorem e1
t:tree
|- fast_flat(t)=slow_flat(t);

THEOREM 1 ACCEPTED


rule +1
t:tree,l:list
|- fast_flat1(t,l)=>append(slow_flat(t),l);

 +1 ACCEPTED


prove +1;

PROVING +1
t:tree,l:list
|- fast_flat1(t,l)=>append(slow_flat(t),l)


induct t +1


NEW GOALS:

[1]
t:tree,l:list
#1 t : atom
&1 [t':tree,l':list
   #1 t' c t
   |- fast_flat1(t',l')=>append(slow_flat(t'),l')]
|- fast_flat1(t,l)=append(slow_flat(t),l)

     SIMPLIFIES TO:

TRUE

[2]
t:tree,l:list
#1 t : join
&1 [t':tree,l':list
   #1 t' c t
   |- fast_flat1(t',l')=>append(slow_flat(t'),l')]
|- fast_flat1(t,l)=append(slow_flat(t),l)

     SIMPLIFIES TO:

t.right:tree,t.left:tree,l:list
```

```
&1 [t':tree,l':list
   #1 t' c join(t.left,t.right)
   |- fast_flat1(t',l')=>append(slow_flat(t'),l')]
|- append(slow_flat(t.left),append(slow_flat(t.right),l))=append(append(
    slow_flat(t.left),slow_flat(t.right)),l)
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
t.right:tree,t.left:tree,l:list
&1 [t':tree,l':list
   #1 t' c join(t.left,t.right)
   |- fast_flat1(t',l')=>append(slow_flat(t'),l')]
|- append(slow_flat(t.left),append(slow_flat(t.right),l))=append(append(
    slow_flat(t.left),slow_flat(t.right)),l)


rule +2
x:list,y:list,z:list
|- append(append(x,y),z)=>append(x,append(y,z))!

 +2 ACCEPTED

CURRENT GOAL:

[1]
t.right:tree,t.left:tree,l:list
&1 [t':tree,l':list
   #1 t' c join(t.left,t.right)
   |- fast_flat1(t',l')=>append(slow_flat(t'),l')]
|- append(slow_flat(t.left),append(slow_flat(t.right),l))=append(append(
    slow_flat(t.left),slow_flat(t.right)),l)

     SIMPLIFIES TO:

TRUE
PROOF OF +1 RELATIVE TO THE ASSERTIONS:
 +2
UNPROVED THEOREMS: e1
UNPROVED RULES:  +2

qed;


prove e1!

PROVING e1
t:tree
|- fast_flat(t)=slow_flat(t)

CURRENT GOAL:

[1]
t:tree
|- fast_flat(t)=slow_flat(t)

     SIMPLIFIES TO:

t:tree
|- append(slow_flat(t),NIL)=slow_flat(t)
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
t:tree
|- append(slow_flat(t),NIL)=slow_flat(t)


rule +3
```

```
x:list
|- append(x,NIL)=>x!

 +3 ACCEPTED

CURRENT GOAL:

[1]
t:tree
|- append(slow_flat(t),NIL)=slow_flat(t)

     SIMPLIFIES TO:

TRUE
PROOF OF #1 RELATIVE TO THE ASSERTIONS:
 +3 +2
UNPROVED RULES:  +3 +2

qed;


prove +2;

PROVING +2
x:list,y:list,z:list
|- append(append(x,y),z)=>append(x,append(y,z))


induct x!


NEW GOALS:

[1]
x:list,y:list,z:list
#1 x : NIL
&1 [x':list,y':list,z':list
   #1 x' c x
   |- append(append(x',y'),z')=append(x',append(y',z')) => TRUE]
|- append(append(x,y),z)=append(x,append(y,z))

     SIMPLIFIES TO:

TRUE

[2]
x:list,y:list,z:list
#1 x : cons
&1 [x':list,y':list,z':list
   #1 x' c x
   |- append(append(x',y'),z')=append(x',append(y',z')) => TRUE]
|- append(append(x,y),z)=append(x,append(y,z))

     SIMPLIFIES TO:

TRUE
PROOF OF +2
UNPROVED RULES:  +3

qed;


prove +3;

PROVING +3
x:list
|- append(x,NIL)=>x
```

```
induct x +!

NEW GOALS:

[1]
x:list
#1 x : NIL
&1 [x':list
   #1 x' c x
   |- append(x',NIL)=>x']
|- append(x,NIL)=x

     SIMPLIFIES TO:

TRUE

[2]
x:list
#1 x : cons
&1 [x':list
   #1 x' c x
   |- append(x',NIL)=>x']
|- append(x,NIL)=x

     SIMPLIFIES TO:

TRUE
PROOF OF +3

qed;
```

A 1.3 Example 3: Total Correctness of Sorting by Merging

```
program sort1;

type list = NIL U cons(car:natnum,cdr:list)
type list_of_cons = NIL U join(hd:cons,tl:list_of_cons)
function drop(l:list):list_of_cons =
  list case l of
    NIL: NIL
    cons: join(cons(car(l),NIL),drop(cdr(l)))
declare function lequal(x:natnum,y:natnum):boolean
function ordered_cons(l:cons):boolean =
  list case cdr(l) of
    NIL: TRUE
    cons: if lequal(car(l),car(cdr(l))) then ordered_cons(cdr(l))
          else FALSE
function ordered(l:list):boolean =
  list case l of
    NIL: TRUE
    cons: ordered_cons(l)
function list_ordered(ll:list_of_cons):boolean =
  list_of_cons case ll of
    NIL: TRUE
    join: if ordered_cons(hd(ll)) then list_ordered(tl(ll))
          else FALSE
declare function pair_merge(ll:list_of_cons):list_of_cons
function sort1(ll:join):list =
  list_of_cons case tl(ll) of
    NIL: hd(ll)
    join: sort1(pair_merge(ll))
function sort(l:list):list =
  list case l of
```

```
    NIL: NIL
    cons: sort1(drop(l))
function length(l1:list_of_cons):natnum =
  list_of_cons case l1 of
    NIL: ZERO
    join: suc(length(tl(l1)))
=
THE FOLLOWING FUNCTIONS ARE UNDEFINED:
  pair_merge
  lequal
THE FOLLOWING SYNTAX EXCEPTIONS HAVE BEEN GENERATED BY THE PARSER:

[α1]
l1:join
#1 tl(l1) : join
|- pair_merge(l1) : join;

[α2]
l:cons
|- drop(l) : join;

THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:

[-1]
l:list
|- drop(l) : list_of_cons;

[-2]
x:natnum,y:natnum
|- lequal(x,y) : boolean;

[-3]
l:cons
|- ordered_cons(l) : boolean;

[-4]
l:list
|- ordered(l) : boolean;

[-5]
l1:list_of_cons
|- list_ordered(l1) : boolean;

[-6]
l1:list_of_cons
|- pair_merge(l1) : list_of_cons;

[-7]
l1:join
|- sort1(l1) : list;

[-8]
l:list
|- sort(l) : list;

[-9]
l1:list_of_cons
|- length(l1) : natnum;


program perm1;
declare function permutation(l1:list, l2:list): boolean
declare function append(l1:list, l2:list): list
function list_append(l:list_of_cons): list =
  list_of_cons case l of
    NIL: NIL
    join: append(hd(l),list_append(tl(l)))
```

```
function list_permutation(l1:list_of_cons, l2:list_of_cons): boolean =
  permutation(list_append(l1),list_append(l2))
=
THE FOLLOWING FUNCTIONS ARE UNDEFINED:
  append
  permutation
  pair_merge
  lequal
THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:

[-10]
l1:list,l2:list
|- permutation(l1,l2) : boolean;

[-11]
l1:list,l2:list
|- append(l1,l2) : list;

[-12]
l:list_of_cons
|- list_append(l) : list;

[-13]
l1:list_of_cons,l2:list_of_cons
|- list_permutation(l1,l2) : boolean;


theorem e1
x:list
|- ordered(sort(x));
THEOREM 1 ACCEPTED


theorem e2
x:list
|- permutation(sort(x),x)=TRUE;
THEOREM 2 ACCEPTED


rule +1
y:join
#1 suc(ZERO) c length(y)
|- length(pair_merge(y)) c length(y)=>TRUE;
 +1 ACCEPTED


rule +2
l1:join
|- pair_merge(l1) : join;
 +2 ACCEPTED


rule +3
y:join
#1 list_ordered(y)
|- ordered(sortl(y))=>TRUE;
 +3 ACCEPTED


prove e1;
PROVING e1
x:list
|- ordered(sort(x))


type x!
```

```
NEW GOALS:

[1]
x:list
#1 x : NIL
|- ordered(sort(x))

     SIMPLIFIES TO:

TRUE

[2]
x:list
#1 x : cons
|- ordered(sort(x))

     SIMPLIFIES TO:

x.cdr:list,x.car:natnum
|- ordered(sort1(join(cons(x.car,NIL),drop(x.cdr))))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.cdr:list,x.car:natnum
|- ordered(sort1(join(cons(x.car,NIL),drop(x.cdr))))


consequence !list_ordered(join(cons(x.car,NIL),drop(x.cdr)))!

NEW GOALS:

[1]
x.cdr:list,x.car:natnum
#1 list_ordered(join(cons(x.car,NIL),drop(x.cdr)))
|- ordered(sort1(join(cons(x.car,NIL),drop(x.cdr))))

     SIMPLIFIES TO:

TRUE

[2]
x.cdr:list,x.car:natnum
|- list_ordered(join(cons(x.car,NIL),drop(x.cdr)))

     SIMPLIFIES TO:

x.cdr:list,x.car:natnum
|- list_ordered(drop(x.cdr))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.cdr:list,x.car:natnum
|- list_ordered(drop(x.cdr))


induct x.cdr!

NEW GOALS:

[1]
x.cdr:list,x.car:natnum
#1 x.cdr : NIL
&1 [x.cdr':list,x.car':natnum
   #1 x.cdr' c x.cdr
   |- list_ordered(drop(x.cdr'))=>TRUE]
|- list_ordered(drop(x.cdr))
```

```
     SIMPLIFIES TO:

TRUE

[2]
x.cdr:list,x.car:natnum
#1 x.cdr : cons
&1 [x.cdr':list,x.car':natnum
   #1 x.cdr' c x.cdr
   |- list_ordered(drop(x.cdr'))=>TRUE]
|- list_ordered(drop(x.cdr))

     SIMPLIFIES TO:

TRUE
PROOF OF #1 RELATIVE TO THE ASSERTIONS:
 +3
UNPROVED THEOREMS: #2
UNPROVED RULES:  +3
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

qed;

rule +4
n.pred:natnum
|- ZERO c suc(n.pred)=>TRUE;
 +4 ACCEPTED


rule +5
y:list_of_cons
#1 list_ordered(y)
|- list_ordered(pair_merge(y))=>TRUE;
 +5 ACCEPTED


rule +6
p:boolean,x:boolean
|- if p then x else FALSE=>p and x;
 +6 ACCEPTED


prove +3;
PROVING +3
y:join
#1 list_ordered(y)
|- ordered(sort1(y))=>TRUE


induct length(y) +1

NEW GOALS:

[1]
y:join
#1 length(y) : ZERO
#2 list_ordered(y)
&1 [y':join
   #1 length(y') c length(y)
   #2 list_ordered(y')
   |- ordered(sort1(y'))=>TRUE]
|- ordered(sort1(y))=TRUE

     SIMPLIFIES TO:

TRUE
```

```
[2]
y:join
#1 length(y) : suc
#2 list_ordered(y)
&1 [y':join
   #1 length(y') c length(y)
   #2 list_ordered(y')
   |- ordered(sort1(y'))=>TRUE]
|- ordered(sort1(y))=TRUE

     SIMPLIFIES TO:

y.t1:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 ordered_cons(cons(y.hd.car,y.hd.cdr))
#2 list_ordered(y.t1)
&1 [y':join
   #1 length(y') c length(join(cons(y.hd.car,y.hd.cdr),y.t1))
   #2 list_ordered(y')
   |- ordered(sort1(y'))=>TRUE]
|- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.t1)))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
y.t1:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 ordered_cons(cons(y.hd.car,y.hd.cdr))
#2 list_ordered(y.t1)
&1 [y':join
   #1 length(y') c length(join(cons(y.hd.car,y.hd.cdr),y.t1))
   #2 list_ordered(y')
   |- ordered(sort1(y'))=>TRUE]
|- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.t1)))


type y.t1!

NEW GOALS:

[1]
y.t1:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 y.t1 : NIL
#2 ordered_cons(cons(y.hd.car,y.hd.cdr))
#3 list_ordered(y.t1)
&1 [y':join
   #1 length(y') c length(join(cons(y.hd.car,y.hd.cdr),y.t1))
   #2 list_ordered(y')
   |- ordered(sort1(y'))=>TRUE]
|- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.t1)))

     SIMPLIFIES TO:

TRUE

[2]
y.t1:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 y.t1 : join
#2 ordered_cons(cons(y.hd.car,y.hd.cdr))
#3 list_ordered(y.t1)
&1 [y':join
   #1 length(y') c length(join(cons(y.hd.car,y.hd.cdr),y.t1))
   #2 list_ordered(y')
   |- ordered(sort1(y'))=>TRUE]
|- ordered(sort1(join(cons(y.hd.car,y.hd.cdr),y.t1)))

     SIMPLIFIES TO:

TRUE
PROOF OF +3 RELATIVE TO THE ASSERTIONS:
```

```
 +2 +4 +1 +5 +6
UNPROVED THEOREMS: e2
UNPROVED RULES:  +6 +5 +4 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

qed;

prove +4;
PROVING +4
n.pred:natnum
|- ZERO c suc(n.pred)=>TRUE


induct n.pred!

NEW GOALS:

[1]
n.pred:natnum
#1 n.pred : ZERO
&1 [n.pred':natnum
   #1 n.pred' c n.pred
   |- ZERO c suc(n.pred')=>TRUE]
|- ZERO c suc(n.pred)=TRUE

     SIMPLIFIES TO:

TRUE

[2]
n.pred:natnum
#1 n.pred : suc
&1 [n.pred':natnum
   #1 n.pred' c n.pred
   |- ZERO c suc(n.pred')=>TRUE]
|- ZERO c suc(n.pred)=TRUE

     SIMPLIFIES TO:

TRUE
PROOF OF +4
UNPROVED THEOREMS: e2
UNPROVED RULES:  +6 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

qed;

prove +6;
PROVING +6
p:boolean,x:boolean
|- if p then x else FALSE=>p and x


type p!

NEW GOALS:

[1]
p:boolean,x:boolean
#1 p : TRUE
|- if p then x else FALSE=p and x

     SIMPLIFIES TO:

TRUE

[2]
```

```
p:boolean,x:boolean
#1 p : FALSE
|- if p then x else FALSE=p and x

     SIMPLIFIES TO:

TRUE
PROOF OF +6
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

qed;

prove -1;
PROVING -1
l:list
|- drop(l) : list_of_cons


induct ll

NEW GOALS:

[1]
l:list
#1 l : NIL
&1 [l':list
   #1 l' c l
   |- drop(l') : list_of_cons]
|- drop(l) : list_of_cons

     SIMPLIFIES TO:

TRUE

[2]
l:list
#1 l : cons
&1 [l':list
   #1 l' c l
   |- drop(l') : list_of_cons]
|- drop(l) : list_of_cons

     SIMPLIFIES TO:

TRUE
PROOF OF -1
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2

qed;
prove -3;
PROVING -3
l:cons
|- ordered_cons(l) : boolean


induct ll

NEW GOALS:

[1]
l:cons
&1 [l':cons
   #1 l' c l
```

```
   |- ordered_cons(l') : boolean]
|- ordered_cons(l) : boolean

     SIMPLIFIES TO:

l.cdr:list,l.car:natnum,l:cons
&1 [l':cons
   #1 l' = cons(l.car,l.cdr)
   |- ordered_cons(l') : boolean]
|- ordered_cons(cons(l.car,l.cdr)) : boolean
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
l.cdr:list,l.car:natnum,l:cons
&1 [l':cons
   #1 l' = cons(l.car,l.cdr)
   |- ordered_cons(l') : boolean]
|- ordered_cons(cons(l.car,l.cdr)) : boolean


type l.cdr!

NEW GOALS:

[1]
l.cdr:list,l.car:natnum,l:cons
#1 l.cdr : NIL
&1 [l':cons
   #1 l' = cons(l.car,l.cdr)
   |- ordered_cons(l') : boolean]
|- ordered_cons(cons(l.car,l.cdr)) : boolean

     SIMPLIFIES TO:

TRUE

[2]
l.cdr:list,l.car:natnum,l:cons
#1 l.cdr : cons
&1 [l':cons
   #1 l' = cons(l.car,l.cdr)
   |- ordered_cons(l') : boolean]
|- ordered_cons(cons(l.car,l.cdr)) : boolean

     SIMPLIFIES TO:

TRUE
PROOF OF -3
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -2

qed;

prove -4;
PROVING -4
l:list
|- ordered(l) : boolean


type l!

NEW GOALS:

[1]
l:list
#1 l : NIL
```

```
|- ordered(l) : boolean

     SIMPLIFIES TO:

TRUE

[2]
l:list
#1 l : cons
|- ordered(l) : boolean

     SIMPLIFIES TO:

TRUE
PROOF OF -4
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -5 -2

qed;

prove -5;
PROVING -5
ll:list_of_cons
|- list_ordered(ll) : boolean


induct lll

NEW GOALS:

[1]
ll:list_of_cons
#1 ll : NIL
&1 [ll':list_of_cons
   #1 ll' c ll
   |- list_ordered(ll') : boolean]
|- list_ordered(ll) : boolean

     SIMPLIFIES TO:

TRUE

[2]
ll:list_of_cons
#1 ll : join
&1 [ll':list_of_cons
   #1 ll' c ll
   |- list_ordered(ll') : boolean]
|- list_ordered(ll) : boolean

     SIMPLIFIES TO:

TRUE
PROOF OF -5
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -7 -6 -2

qed;

prove -7;
PROVING -7
ll:join
|- sort1(ll) : list
```

```
induct length(l1)!

NEW GOALS:

[1]
l1:join
#1 length(l1) : ZERO
&1 [l1':join
   #1 length(l1') c length(l1)
   |- sort1(l1') : list]
|- sort1(l1) : list

     SIMPLIFIES TO:

TRUE

[2]
l1:join
#1 length(l1) : suc
&1 [l1':join
   #1 length(l1') c length(l1)
   |- sort1(l1') : list]
|- sort1(l1) : list

     SIMPLIFIES TO:

l1.t1:list_of_cons,l1.hd.cdr:list,l1.hd.car:natnum,l1:join
&1 [l1':join
   #1 length(l1') c length(join(cons(l1.hd.car,l1.hd.cdr),l1.t1))
   |- sort1(l1') : list]
|- sort1(join(cons(l1.hd.car,l1.hd.cdr),l1.t1)) : list
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
l1.t1:list_of_cons,l1.hd.cdr:list,l1.hd.car:natnum,l1:join
&1 [l1':join
   #1 length(l1') c length(join(cons(l1.hd.car,l1.hd.cdr),l1.t1))
   |- sort1(l1') : list]
|- sort1(join(cons(l1.hd.car,l1.hd.cdr),l1.t1)) : list


type l1.t1!

NEW GOALS:

[1]
l1.t1:list_of_cons,l1.hd.cdr:list,l1.hd.car:natnum,l1:join
#1 l1.t1 : NIL
&1 [l1':join
   #1 length(l1') c length(join(cons(l1.hd.car,l1.hd.cdr),l1.t1))
   |- sort1(l1') : list]
|- sort1(join(cons(l1.hd.car,l1.hd.cdr),l1.t1)) : list

     SIMPLIFIES TO:

TRUE

[2]
l1.t1:list_of_cons,l1.hd.cdr:list,l1.hd.car:natnum,l1:join
#1 l1.t1 : join
&1 [l1':join
   #1 length(l1') c length(join(cons(l1.hd.car,l1.hd.cdr),l1.t1))
   |- sort1(l1') : list]
|- sort1(join(cons(l1.hd.car,l1.hd.cdr),l1.t1)) : list

     SIMPLIFIES TO:
```

```
TRUE
PROOF OF -7 RELATIVE TO THE ASSERTIONS:
 +2 +1
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -8 -6 -2

qed;

prove -8;
PROVING -8
l:list
|- sort(l) : list


type ll

NEW GOALS:

[1]
l:list
#1 l : NIL
|- sort(l) : list

     SIMPLIFIES TO:

TRUE

[2]
l:list
#1 l : cons
|- sort(l) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -8
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -9 -6 -2

qed;

prove -9;
PROVING -9
ll:list_of_cons
|- length(ll) : natnum


induct lll

NEW GOALS:

[1]
ll:list_of_cons
#1 ll : NIL
&1 [ll':list_of_cons
   #1 ll' c ll
   |- length(ll') : natnum]
|- length(ll) : natnum

     SIMPLIFIES TO:

TRUE

[2]
ll:list_of_cons
```

```
#1 ll : join
&1 [ll':list_of_cons
   #1 ll' c ll
   |- length(ll') : natnum]
|- length(ll) : natnum

     SIMPLIFIES TO:

TRUE
PROOF OF -9
UNPROVED THEOREMS: e2
UNPROVED RULES:  +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -6 -2

qed;

rule +7
x:list
|- permutation(x,x)=>TRUE;
 +7 ACCEPTED


rule +8
x:join,y:list
#1 list_permutation(x,drop(y))
|- permutation(sort1(x),y)=>TRUE;
 +8 ACCEPTED


prove e2;
PROVING e2
x:list
|- permutation(sort(x),x)=TRUE


type x1

NEW GOALS:

[1]
x:list
#1 x : NIL
|- permutation(sort(x),x)=TRUE

     SIMPLIFIES TO:

TRUE

[2]
x:list
#1 x : cons
|- permutation(sort(x),x)=TRUE

     SIMPLIFIES TO:

TRUE
PROOF OF e2 RELATIVE TO THE ASSERTIONS:
 +7 +8
UNPROVED RULES:  +8 +7 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -6 -2

qed;

prove +8;
PROVING +8
x:join,y:list
#1 list_permutation(x,drop(y))
```

```
|- permutation(sort1(x),y)=>TRUE


rule +9
x:list_of_cons,y:list_of_cons
#1 list_permutation(x,y)
|- list_permutation(pair_merge(x),y)=>TRUE;
 +9 ACCEPTED


rule +10
x:list
|- append(x,NIL)=>x;
 +10 ACCEPTED


rule +11
x:list
|- list_append(drop(x))=>x;
 +11 ACCEPTED


induct length(x) +1

NEW GOALS:

[1]
x:join,y:list
#1 length(x) : ZERO
#2 list_permutation(x,drop(y))
&1 [x':join,y':list
   #1 length(x') < length(x)
   #2 list_permutation(x',drop(y'))
   |- permutation(sort1(x'),y')=>TRUE]
|- permutation(sort1(x),y)=TRUE

     SIMPLIFIES TO:

TRUE

[2]
x:join,y:list
#1 length(x) : suc
#2 list_permutation(x,drop(y))
&1 [x':join,y':list
   #1 length(x') < length(x)
   #2 list_permutation(x',drop(y'))
   |- permutation(sort1(x'),y')=>TRUE]
|- permutation(sort1(x),y)=TRUE

     SIMPLIFIES TO:

x.t1:list_of_cons,x.hd.cdr:list,x.hd.car:natnum,x:join,y:list
#1 permutation(append(cons(x.hd.car,x.hd.cdr),list_append(x.t1)),y)
&1 [x':join,y':list
   #1 length(x') < length(join(cons(x.hd.car,x.hd.cdr),x.t1))
   #2 list_permutation(x',drop(y'))
   |- permutation(sort1(x'),y')=>TRUE]
|- permutation(sort1(join(cons(x.hd.car,x.hd.cdr),x.t1)),y)
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.t1:list_of_cons,x.hd.cdr:list,x.hd.car:natnum,x:join,y:list
#1 permutation(append(cons(x.hd.car,x.hd.cdr),list_append(x.t1)),y)
&1 [x':join,y':list
   #1 length(x') < length(join(cons(x.hd.car,x.hd.cdr),x.t1))
   #2 list_permutation(x',drop(y'))
```

```
   |- permutation(sort1(x'),y')=>TRUE]
|- permutation(sort1(join(cons(x.hd.car,x.hd.cdr),x.tl)),y)


type x.tl!

NEW GOALS:

[1]
x.tl:list_of_cons,x.hd.cdr:list,x.hd.car:natnum,x:join,y:list
#1 x.tl : NIL
#2 permutation(append(cons(x.hd.car,x.hd.cdr),list_append(x.tl)),y)
&1 [x':join,y':list
   #1 length(x') c length(join(cons(x.hd.car,x.hd.cdr),x.tl))
   #2 list_permutation(x',drop(y'))
   |- permutation(sort1(x'),y')=>TRUE]
|- permutation(sort1(join(cons(x.hd.car,x.hd.cdr),x.tl)),y)

     SIMPLIFIES TO:

TRUE

[2]
x.tl:list_of_cons,x.hd.cdr:list,x.hd.car:natnum,x:join,y:list
#1 x.tl : join
#2 permutation(append(cons(x.hd.car,x.hd.cdr),list_append(x.tl)),y)
&1 [x':join,y':list
   #1 length(x') c length(join(cons(x.hd.car,x.hd.cdr),x.tl))
   #2 list_permutation(x',drop(y'))
   |- permutation(sort1(x'),y')=>TRUE]
|- permutation(sort1(join(cons(x.hd.car,x.hd.cdr),x.tl)),y)

     SIMPLIFIES TO:

TRUE
PROOF OF +8 RELATIVE TO THE ASSERTIONS:
 +10 +2 +1 +9 +11
UNPROVED RULES:  +11 +10 +9 +7 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -8 -2

qed;
rule +12
x:list,y:list,z:list
%1 permutation(y,z)
#2 permutation(x,y)
|- permutation(x,z)=>TRUE;
 +12 ACCEPTED


rule +13
x:list_of_cons
|- permutation(list_append(pair_merge(x)),list_append(x))=>TRUE;
 +13 ACCEPTED


prove +9!
PROVING +9
x:list_of_cons,y:list_of_cons
#1 list_permutation(x,y)
|- list_permutation(pair_merge(x),y)=>TRUE

CURRENT GOAL:

[1]
x:list_of_cons,y:list_of_cons
#1 list_permutation(x,y)
|- list_permutation(pair_merge(x),y)=TRUE
```

```
     SIMPLIFIES TO:

TRUE
PROOF OF +9 RELATIVE TO THE ASSERTIONS:
 +13 +12
UNPROVED RULES:  +13 +12 +11 +10 +7 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -12 -11 -10 -8 -2

qed;

prove -12;
PROVING -12
l:list_of_cons
|- list_append(l) : list


induct l

NEW GOALS:

[1]
l:list_of_cons
#1 l : NIL
&1 [l':list_of_cons
   #1 l' c l
   |- list_append(l') : list]
|- list_append(l) : list

     SIMPLIFIES TO:

TRUE

[2]
l:list_of_cons
#1 l : join
&1 [l':list_of_cons
   #1 l' c l
   |- list_append(l') : list]
|- list_append(l) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -12
UNPROVED RULES:  +13 +12 +11 +10 +7 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -13 -11 -10 -8 -2

qed;

prove -13!
PROVING -13
l1:list_of_cons,l2:list_of_cons
|- list_permutation(l1,l2) : boolean

CURRENT GOAL:

[1]
(l1:list_of_cons,l2:list_of_cons
|- list_permutation(l1,l2) : boolean

     SIMPLIFIES TO:

TRUE
PROOF OF -13
UNPROVED RULES:  +13 +12 +11 +10 +7 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -11 -10 -8 -2
```

```
qed;

program sort2;
declare function merge_cons(l1:cons,l2:cons):cons
function pair_merge(ll:list_of_cons):list_of_cons s
  list_of_cons case ll of
    NIL: NIL
    join: list_of_cons case tl(ll) of
            NIL: ll
            join: join(merge_cons(hd(ll),hd(tl(ll))),pair_merge(tl(tl(ll))))
*
THE FOLLOWING FUNCTIONS ARE UNDEFINED:
  merge_cons
  append
  permutation
  lequal
THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:

[-14]
l1:cons,l2:cons
|- merge_cons(l1,l2) : cons;


rule +14
x:list,y:list,z:list_of_cons
|- append(x,append(y,list_append(z)))=>append(append(x,y),list_append(z));
 +14 ACCEPTED


rule +15
x:cons,y:cons
|- permutation(merge_cons(x,y),append(x,y))=>TRUE;
 +15 ACCEPTED


rule +16
x:list,y:list,u:list,v:list
#1 permutation(x,y)
#2 permutation(u,v)
|- permutation(append(x,u),append(y,v))=>TRUE;
 +16 ACCEPTED


rule +17
x:list
|- append(NIL,x)=>x;
 +17 ACCEPTED


prove +13;
PROVING +13
x:list_of_cons
|- permutation(list_append(pair_merge(x)),list_append(x))=>TRUE


induct x!

NEW GOALS:

[1]
x:list_of_cons
#1 x : NIL
&1 [x':list_of_cons
   #1 x' c x
   |- permutation(list_append(pair_merge(x')),list_append(x'))=>TRUE]
|- permutation(list_append(pair_merge(x)),list_append(x))=TRUE
```

```
     SIMPLIFIES TO:

TRUE

[2]
x:list_of_cons
#1 x : join
&1 [x':list_of_cons
   #1 x' c x
   |- permutation(list_append(pair_merge(x')),list_append(x'))=>TRUE]
|- permutation(list_append(pair_merge(x)),list_append(x))=TRUE

     SIMPLIFIES TO:

x.tl:list_of_cons,x.hd.cdr:list,x.hd.car:natnum
&1 [x':list_of_cons
   #1 x' c join(cons(x.hd.car,x.hd.cdr),x.tl)
   |- permutation(list_append(pair_merge(x')),list_append(x'))=>TRUE]
|- permutation(append(hd(list_of_cons case of x.tl { NIL: join(cons(x.hd.car,
    x.hd.cdr),NIL)  join: join(merge_cons(cons(x.hd.car,x.hd.cdr),cons(
    x.tl.hd.car,x.tl.hd.cdr)),pair_merge(x.tl.tl))}),list_append(tl(list_of_cons
     case of x.tl { NIL: join(cons(x.hd.car,x.hd.cdr),NIL)  join: join(
    merge_cons(cons(x.hd.car,x.hd.cdr),cons(x.tl.hd.car,x.tl.hd.cdr)),pair_merge
    (x.tl.tl))})})),append(cons(x.hd.car,x.hd.cdr),list_append(x.tl)))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.tl:list_of_cons,x.hd.cdr:list,x.hd.car:natnum
&1 [x':list_of_cons
   #1 x' c join(cons(x.hd.car,x.hd.cdr),x.tl)
   |- permutation(list_append(pair_merge(x')),list_append(x'))=>TRUE]
|- permutation(append(hd(list_of_cons case of x.tl { NIL: join(cons(x.hd.car,
    x.hd.cdr),NIL)  join: join(merge_cons(cons(x.hd.car,x.hd.cdr),cons(
    x.tl.hd.car,x.tl.hd.cdr)),pair_merge(x.tl.tl))}),list_append(tl(list_of_cons
     case of x.tl { NIL: join(cons(x.hd.car,x.hd.cdr),NIL)  join: join(
    merge_cons(cons(x.hd.car,x.hd.cdr),cons(x.tl.hd.car,x.tl.hd.cdr)),pair_merge
    (x.tl.tl))})})),append(cons(x.hd.car,x.hd.cdr),list_append(x.tl)))


type x.tl!

NEW GOALS:

[1]
x.tl:list_of_cons,x.hd.cdr:list,x.hd.car:natnum
#1 x.tl : NIL
&1 [x':list_of_cons
   #1 x' c join(cons(x.hd.car,x.hd.cdr),x.tl)
   |- permutation(list_append(pair_merge(x')),list_append(x'))=>TRUE]
|- permutation(append(hd(list_of_cons case of x.tl { NIL: join(cons(x.hd.car,
    x.hd.cdr),NIL)  join: join(merge_cons(cons(x.hd.car,x.hd.cdr),cons(
    x.tl.hd.car,x.tl.hd.cdr)),pair_merge(x.tl.tl))}),list_append(tl(list_of_cons
     case of x.tl { NIL: join(cons(x.hd.car,x.hd.cdr),NIL)  join: join(
    merge_cons(cons(x.hd.car,x.hd.cdr),cons(x.tl.hd.car,x.tl.hd.cdr)),pair_merge
    (x.tl.tl))})})),append(cons(x.hd.car,x.hd.cdr),list_append(x.tl)))

     SIMPLIFIES TO:

TRUE

[2]
x.tl:list_of_cons,x.hd.cdr:list,x.hd.car:natnum
#1 x.tl : join
&1 [x':list_of_cons
   #1 x' c join(cons(x.hd.car,x.hd.cdr),x.tl)
   |- permutation(list_append(pair_merge(x')),list_append(x'))=>TRUE]
```

```
|- permutation(append(hd(list_of_cons case of x.tl ! NIL: join(cons(x.hd.car,
    x.hd.cdr),NIL)  join: join(merge_cons(cons(x.hd.car,x.hd.cdr),cons(
    x.tl.hd.car,x.tl.hd.cdr)),pair_merge(x.tl.tl))!),list_append(tl(list_of_cons
     case of x.tl ! NIL: join(cons(x.hd.car,x.hd.cdr),NIL)  join: join(
    merge_cons(cons(x.hd.car,x.hd.cdr),cons(x.tl.hd.car,x.tl.hd.cdr)),pair_merge
    (x.tl.tl))!))),append(cons(x.hd.car,x.hd.cdr),list_append(x.tl)))

     SIMPLIFIES TO:

TRUE
PROOF OF +13 RELATIVE TO THE ASSERTIONS:
 +16 +14 +15 +7 +2
UNPROVED RULES:  +16 +15 +14 +12 +11 +10 +7 +5 +2 +1
UNPROVED SYNTAX LEMMAS:  -14 -11 -10 -6 -2

qed;

program sort3;
function merge(l1:list,l2:list):list =
  list case l1 of
    NIL: l2
    cons: list case l2 of
            NIL: l1
            cons: merge_cons(l1,l2)
function merge_cons(l1:cons,l2:cons):cons =
  if lequal(car(l1),car(l2)) then cons(car(l1),merge(cdr(l1),l2))
  else cons(car(l2),merge(l1,cdr(l2)))
declare function sum_length(l1:list,l2:list):natnum
=
THE FOLLOWING FUNCTIONS ARE UNDEFINED:
  sum_length
  append
  permutation
  lequal
THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:

[-15]
l1:list,l2:list
|- merge(l1,l2) : list;

[-16]
l1:list,l2:list
|- sum_length(l1,l2) : natnum;


rule +18
x1:cons,x2:cons
#1 ordered_cons(x1)
#2 ordered_cons(x2)
|- ordered_cons(merge_cons(x1,x2))=>TRUE;
 +18 ACCEPTED


prove +5;
PROVING +5
y:list_of_cons
#1 list_ordered(y)
|- list_ordered(pair_merge(y))=>TRUE


induct y +i

NEW GOALS:

[i]
y:list_of_cons
#1 y : NIL
```

```
#2 list_ordered(y)
&1 [y':list_of_cons
   #1 y' c y
   #2 list_ordered(y')
   |- list_ordered(pair_merge(y'))=>TRUE]
|- list_ordered(pair_merge(y))=TRUE

     SIMPLIFIES TO:

TRUE

[2]
y:list_of_cons
#1 y : join
#2 list_ordered(y)
&1 [y':list_of_cons
   #1 y' c y
   #2 list_ordered(y')
   |- list_ordered(pair_merge(y'))=>TRUE]
|- list_ordered(pair_merge(y))=TRUE

     SIMPLIFIES TO:

y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
#1 ordered_cons(cons(y.hd.car,y.hd.cdr))
#2 list_ordered(y.tl)
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 list_ordered(y')
   |- list_ordered(pair_merge(y'))=>TRUE]
|- ordered_cons(hd(list_of_cons case of y.tl ! NIL: join(cons(y.hd.car,y.hd.cdr)
    ,NIL)  join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,
    y.tl.hd.cdr)),pair_merge(y.tl.tl))])) and list_ordered(tl(list_of_cons case
    of y.tl ! NIL: join(cons(y.hd.car,y.hd.cdr),NIL)  join: join(merge_cons(cons
    (y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr)),pair_merge(y.tl.tl))]))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
#1 ordered_cons(cons(y.hd.car,y.hd.cdr))
#2 list_ordered(y.tl)
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 list_ordered(y')
   |- list_ordered(pair_merge(y'))=>TRUE]
|- ordered_cons(hd(list_of_cons case of y.tl ! NIL: join(cons(y.hd.car,y.hd.cdr)
    ,NIL)  join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,
    y.tl.hd.cdr)),pair_merge(y.tl.tl))])) and list_ordered(tl(list_of_cons case
    of y.tl ! NIL: join(cons(y.hd.car,y.hd.cdr),NIL)  join: join(merge_cons(cons
    (y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr)),pair_merge(y.tl.tl))]))


type y.tl!

NEW GOALS:

[1]
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
#1 y.tl : NIL
#2 ordered_cons(cons(y.hd.car,y.hd.cdr))
#3 list_ordered(y.tl)
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 list_ordered(y')
   |- list_ordered(pair_merge(y'))=>TRUE]
|- ordered_cons(hd(list_of_cons case of y.tl ! NIL: join(cons(y.hd.car,y.hd.cdr)
    ,NIL)  join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,
```

```
    y.tl.hd.cdr)),pair_merge(y.tl.tl))])) and list_ordered(tl(list_of_cons case
    of y.tl { NIL: join(cons(y.hd.car,y.hd.cdr),NIL)  join: join(merge_cons(cons
    (y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr)),pair_merge(y.tl.tl))]))

     SIMPLIFIES TO:

TRUE

[2]
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
#1 y.tl : join
#2 ordered_cons(cons(y.hd.car,y.hd.cdr))
#3 list_ordered(y.tl)
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 list_ordered(y')
   |- list_ordered(pair_merge(y'))=>TRUE]
|- ordered_cons(hd(list_of_cons case of y.tl { NIL: join(cons(y.hd.car,y.hd.cdr)
    ,NIL)  join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,
    y.tl.hd.cdr)),pair_merge(y.tl.tl))])) and list_ordered(tl(list_of_cons case
    of y.tl { NIL: join(cons(y.hd.car,y.hd.cdr),NIL)  join: join(merge_cons(cons
    (y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr)),pair_merge(y.tl.tl))]))

     SIMPLIFIES TO:

TRUE
PROOF OF +5 RELATIVE TO THE ASSERTIONS:
 +18 +2
UNPROVED RULES:  +18 +16 +15 +14 +12 +11 +10 +7 +2 +1
UNPROVED SYNTAX LEMMAS:  -16 -15 -14 -11 -10 -6 -2

qed;
prove -6;
PROVING -6
ll:list_of_cons
|- pair_merge(ll) : list_of_cons


induct ll!

NEW GOALS:

[1]
ll:list_of_cons
#1 ll : NIL
&1 [ll':list_of_cons
   #1 ll' c ll
   |- pair_merge(ll') : list_of_cons]
|- pair_merge(ll) : list_of_cons

     SIMPLIFIES TO:

TRUE

[2]
ll:list_of_cons
#1 ll : join
&1 [ll':list_of_cons
   #1 ll' c ll
   |- pair_merge(ll') : list_of_cons]
|- pair_merge(ll) : list_of_cons

     SIMPLIFIES TO:

TRUE
PROOF OF -6 RELATIVE TO THE ASSERTIONS:
 +2
```

```
UNPROVED RULES:  +18 +16 +15 +14 +12 +11 +10 +7 +2 +1
UNPROVED SYNTAX LEMMAS:  -16 -15 -14 -11 -10 -2

qed;

rule +19
x:list,y:list,z:list
#1 x c y
|- sum_length(x,z) c sum_length(y,z)=>TRUE;
 +19 ACCEPTED


rule +20
x:list,y:list,z:list
#1 y c z
|- sum_length(x,y) c sum_length(x,z)=>TRUE;
 +20 ACCEPTED


rule +21
x:cons,y:list
|- sum_length(x,y) : suc;
 +21 ACCEPTED


prove -14;
PROVING -14
l1:cons,l2:cons
|- merge_cons(l1,l2) : cons


induct sum_length(l1,l2)!

NEW GOALS:

[1]
l1:cons,l2:cons
&1 [l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(l1,l2)
   |- merge_cons(l1',l2') : cons]
|- merge_cons(l1,l2) : cons

     SIMPLIFIES TO:

l1.cdr:list,l1.car:natnum,l1:cons,l2.cdr:list,l2.car:natnum,l2:cons
&1 [l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge_cons(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr)) : cons
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
l1.cdr:list,l1.car:natnum,l1:cons,l2.cdr:list,l2.car:natnum,l2:cons
&1 [l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge_cons(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr)) : cons


type !equal(l1.car,l2.car)!

NEW GOALS:

[1]
l1.cdr:list,l1.car:natnum,l1:cons,l2.cdr:list,l2.car:natnum,l2:cons
#1 !equal(l1.car,l2.car) : TRUE
&1 [l1':cons,l2':cons
```

```
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge_cons(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr)) : cons

     SIMPLIFIES TO:

l1.cdr:list,l1.car:natnum,l2.cdr:list,l2.car:natnum
#1 lequal(l1.car,l2.car)
&1 [[l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge(l1.cdr,cons(l2.car,l2.cdr)) : list

[2]
l1.cdr:list,l1.car:natnum,l1:cons,l2.cdr:list,l2.car:natnum,l2:cons
#1 lequal(l1.car,l2.car) : FALSE
&1 [[l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge_cons(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr)) : cons

     SIMPLIFIES TO:

TRUE
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
l1.cdr:list,l1.car:natnum,l2.cdr:list,l2.car:natnum
#1 lequal(l1.car,l2.car)
&1 [[l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge(l1.cdr,cons(l2.car,l2.cdr)) : list


type !l1.cdr!

NEW GOALS:

[1]
l1.cdr:list,l1.car:natnum,l2.cdr:list,l2.car:natnum
#1 l1.cdr : NIL
#2 lequal(l1.car,l2.car)
&1 [[l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge(l1.cdr,cons(l2.car,l2.cdr)) : list

     SIMPLIFIES TO:

TRUE

[2]
l1.cdr:list,l1.car:natnum,l2.cdr:list,l2.car:natnum
#1 l1.cdr : cons
#2 lequal(l1.car,l2.car)
&1 [[l1':cons,l2':cons
   #1 sum_length(l1',l2') c sum_length(cons(l1.car,l1.cdr),cons(l2.car,l2.cdr))
   |- merge_cons(l1',l2') : cons]
|- merge(l1.cdr,cons(l2.car,l2.cdr)) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -14 RELATIVE TO THE ASSERTIONS:
 +19 +20 +21
UNPROVED RULES:  +21 +20 +19 +18 +16 +15 +14 +12 +11 +10 +7 +2 +1
```

```
UNPROVED SYNTAX LEMMAS:  -16 -15 -11 -10 -2

qed;

prove -15;
PROVING -15
l1:list,l2:list
|- merge(l1,l2) : list


type l1

NEW GOALS:

[1]
l1:list,l2:list
#1 l1 : NIL
|- merge(l1,l2) : list

     SIMPLIFIES TO:

TRUE

[2]
l1:list,l2:list
#1 l1 : cons
|- merge(l1,l2) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -15
UNPROVED RULES:  +21 +20 +19 +18 +16 +15 +14 +12 +11 +10 +7 +2 +1
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

rule +22
y:list_of_cons
|- length(pair_merge(y)) c suc(length(y))=>TRUE;
 +22 ACCEPTED


prove +1;
PROVING +1
y:join
#1 suc(ZERO) c length(y)
|- length(pair_merge(y)) c length(y)=>TRUE


induct y1

NEW GOALS:

[1]
y:join
#1 suc(ZERO) c length(y)
&1 [y':join
   #1 y' c y
   #2 suc(ZERO) c length(y')
   |- length(pair_merge(y')) c length(y')=>TRUE]
|- length(pair_merge(y)) c length(y)=TRUE

     SIMPLIFIES TO:

y.t1:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 ZERO c length(y.t1)
```

```
&1 [y':join
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 suc(ZERO) c length(y')
   |- length(pair_merge(y')) c length(y')=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c length(y.tl)
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 ZERO c length(y.tl)
&1 [y':join
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 suc(ZERO) c length(y')
   |- length(pair_merge(y')) c length(y')=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c length(y.tl)


type y.tl!

NEW GOALS:

[1]
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 y.tl : NIL
#2 ZERO c length(y.tl)
&1 [y':join
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 suc(ZERO) c length(y')
   |- length(pair_merge(y')) c length(y')=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c length(y.tl)

    SIMPLIFIES TO:

TRUE

[2]
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum,y:join
#1 y.tl : join
#2 ZERO c length(y.tl)
&1 [y':join
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   #2 suc(ZERO) c length(y')
   |- length(pair_merge(y')) c length(y')=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c length(y.tl)

    SIMPLIFIES TO:

TRUE
PROOF OF +1 RELATIVE TO THE ASSERTIONS:
 +22 +2
UNPROVED RULES:  +22 +21 +20 +19 +18 +16 +15 +14 +12 +11 +10 +7 +2
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

prove +2!
PROVING +2
!1:join
|- pair_merge(!1) : join
```

```
CURRENT GOAL:

[1]
l1:join
|- pair_merge(l1) : join

     SIMPLIFIES TO:

TRUE
PROOF OF +2
UNPROVED RULES:  +22 +21 +20 +19 +18 +16 +15 +14 +12 +11 +10 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;
rule +23
x:natnum,y:natnum
%1 not lequal(x,y)
|- lequal(y,x)=>TRUE;
 +23 ACCEPTED


rule +24
n:natnum,x1:cons,x2:cons
#1 ordered_cons(cons(n,x1))
#2 ordered_cons(cons(n,x2))
|- lequal(n,car(merge_cons(x1,x2)))=>TRUE;
 +24 ACCEPTED


prove +18;
PROVING +18
x1:cons,x2:cons
#1 ordered_cons(x1)
#2 ordered_cons(x2)
|- ordered_cons(merge_cons(x1,x2))=>TRUE


induct sum_length(x1,x2)!

NEW GOALS:

[1]
x1:cons,x2:cons
#1 ordered_cons(x1)
#2 ordered_cons(x2)
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(x1,x2)
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(merge_cons(x1,x2))=TRUE

     SIMPLIFIES TO:

x1.cdr:list,x1.car:natnum,x1:cons,x2.cdr:list,x2.car:natnum,x2:cons
#1 ordered_cons(cons(x1.car,x1.cdr))
#2 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
```

```
x1.cdr:list,x1.car:natnum,x1:cons,x2.cdr:list,x2.car:natnum,x2:cons
#1 ordered_cons(cons(x1.car,x1.cdr))
#2 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))


type !equal(x1.car,x2.car)!

NEW GOALS:

[1]
x1.cdr:list,x1.car:natnum,x1:cons,x2.cdr:list,x2.car:natnum,x2:cons
#1 !equal(x1.car,x2.car) : TRUE
#2 ordered_cons(cons(x1.car,x1.cdr))
#3 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))

     SIMPLIFIES TO:

x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 !equal(x1.car,x2.car)
#2 ordered_cons(cons(x1.car,x1.cdr))
#3 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(cons(x1.car,merge(x1.cdr,cons(x2.car,x2.cdr))))

[2]
x1.cdr:list,x1.car:natnum,x1:cons,x2.cdr:list,x2.car:natnum,x2:cons
#1 !equal(x1.car,x2.car) : FALSE
#2 ordered_cons(cons(x1.car,x1.cdr))
#3 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))

     SIMPLIFIES TO:

x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 not !equal(x1.car,x2.car)
#2 ordered_cons(cons(x1.car,x1.cdr))
#3 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- if !equal(x2.car,car(list case of x2.cdr | NIL: cons(x1.car,x1.cdr)  cons:
    merge_cons(cons(x1.car,x1.cdr),cons(x2.cdr.car,x2.cdr.cdr))])) then
    ordered_cons(cdr(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))) else
    FALSE
```

```
2 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 lequal(x1.car,x2.car)
#2 ordered_cons(cons(x1.car,x1.cdr))
#3 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(cons(x1.car,merge(x1.cdr,cons(x2.car,x2.cdr))))


type x1.cdr!

NEW GOALS:

[1]
x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 x1.cdr : NIL
#2 lequal(x1.car,x2.car)
#3 ordered_cons(cons(x1.car,x1.cdr))
#4 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(cons(x1.car,merge(x1.cdr,cons(x2.car,x2.cdr))))

     SIMPLIFIES TO:

TRUE

[2]
x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 x1.cdr : cons
#2 lequal(x1.car,x2.car)
#3 ordered_cons(cons(x1.car,x1.cdr))
#4 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- ordered_cons(cons(x1.car,merge(x1.cdr,cons(x2.car,x2.cdr))))

     SIMPLIFIES TO:

TRUE
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 not lequal(x1.car,x2.car)
#2 ordered_cons(cons(x1.car,x1.cdr))
#3 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') < sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- if lequal(x2.car,car(list case of x2.cdr [ NIL: cons(x1.car,x1.cdr)  cons:
    merge_cons(cons(x1.car,x1.cdr),cons(x2.cdr.car,x2.cdr.cdr))])) then
    ordered_cons(cdr(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))) else
```

```
    FALSE

type x2.cdr!

NEW GOALS:

[1]
x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 x2.cdr : NIL
#2 not lequal(x1.car,x2.car)
#3 ordered_cons(cons(x1.car,x1.cdr))
#4 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') c sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- if lequal(x2.car,car(list case of x2.cdr | NIL: cons(x1.car,x1.cdr)  cons:
    merge_cons(cons(x1.car,x1.cdr),cons(x2.cdr.car,x2.cdr.cdr))])) then
    ordered_cons(cdr(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))) else
    FALSE

     SIMPLIFIES TO:

TRUE

[2]
x1.cdr:list,x1.car:natnum,x2.cdr:list,x2.car:natnum
#1 x2.cdr : cons
#2 not lequal(x1.car,x2.car)
#3 ordered_cons(cons(x1.car,x1.cdr))
#4 ordered_cons(cons(x2.car,x2.cdr))
&1 [x1':cons,x2':cons
   #1 sum_length(x1',x2') c sum_length(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr))
   #2 ordered_cons(x1')
   #3 ordered_cons(x2')
   |- ordered_cons(merge_cons(x1',x2'))=>TRUE]
|- if lequal(x2.car,car(list case of x2.cdr | NIL: cons(x1.car,x1.cdr)  cons:
    merge_cons(cons(x1.car,x1.cdr),cons(x2.cdr.car,x2.cdr.cdr))])) then
    ordered_cons(cdr(merge_cons(cons(x1.car,x1.cdr),cons(x2.car,x2.cdr)))) else
    FALSE

     SIMPLIFIES TO:

TRUE
PROOF OF +18 RELATIVE TO THE ASSERTIONS:
 +23 +20 +24 +19 +21
UNPROVED RULES:  +24 +23 +22 +21 +20 +19 +16 +15 +14 +12 +11 +10 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

prove +22;
PROVING +22
y:list_of_cons
|- length(pair_merge(y)) c suc(length(y))=>TRUE


induct y!

NEW GOALS:

[1]
y:list_of_cons
#1 y : NIL
&1 [y':list_of_cons
```

```
   #1 y' c y
   |- length(pair_merge(y')) c suc(length(y'))=>TRUE]
|- length(pair_merge(y)) c suc(length(y))=TRUE

     SIMPLIFIES TO:

TRUE

[2]
y:list_of_cons
#1 y : join
&1 [y':list_of_cons
   #1 y' c y
   |- length(pair_merge(y')) c suc(length(y'))=>TRUE]
|- length(pair_merge(y)) c suc(length(y))=TRUE

     SIMPLIFIES TO:

y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   |- length(pair_merge(y')) c suc(length(y'))=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c suc(length(y.tl))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   |- length(pair_merge(y')) c suc(length(y'))=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c suc(length(y.tl))


type y.tl!

NEW GOALS:

[1]
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
#1 y.tl : NIL
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   |- length(pair_merge(y')) c suc(length(y'))=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c suc(length(y.tl))

     SIMPLIFIES TO:

TRUE

[2]
y.tl:list_of_cons,y.hd.cdr:list,y.hd.car:natnum
#1 y.tl : join
&1 [y':list_of_cons
   #1 y' c join(cons(y.hd.car,y.hd.cdr),y.tl)
   |- length(pair_merge(y')) c suc(length(y'))=>TRUE]
|- length(tl(list_of_cons case of y.tl | NIL: join(cons(y.hd.car,y.hd.cdr),NIL)
      join: join(merge_cons(cons(y.hd.car,y.hd.cdr),cons(y.tl.hd.car,y.tl.hd.cdr
   )),pair_merge(y.tl.tl))))) c suc(length(y.tl))

     SIMPLIFIES TO:
```

```
TRUE
PROOF OF +22
UNPROVED RULES:  +24 +23 +21 +20 +19 +16 +15 +14 +12 +11 +10 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;
prove +24;
PROVING +24
n:natnum,x1:cons,x2:cons
#1 ordered_cons(cons(n,x1))
#2 ordered_cons(cons(n,x2))
|- lequal(n,car(merge_cons(x1,x2)))=>TRUE


type lequal(car(x1),car(x2))!

NEW GOALS:

[1]
n:natnum,x1:cons,x2:cons
#1 lequal(car(x1),car(x2)) : TRUE
#2 ordered_cons(cons(n,x1))
#3 ordered_cons(cons(n,x2))
|- lequal(n,car(merge_cons(x1,x2)))=TRUE

     SIMPLIFIES TO:

TRUE

[2]
n:natnum,x1:cons,x2:cons
#1 lequal(car(x1),car(x2)) : FALSE
#2 ordered_cons(cons(n,x1))
#3 ordered_cons(cons(n,x2))
|- lequal(n,car(merge_cons(x1,x2)))=TRUE

     SIMPLIFIES TO:

TRUE
PROOF OF +24
UNPROVED RULES:  +23 +21 +20 +19 +16 +15 +14 +12 +11 +10 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

program perm2;
function append(l1:list, l2:list): list =
  list case l1 of
    NIL: l2
    cons: cons(car(l1),append(cdr(l1),l2))
=
THE FOLLOWING FUNCTIONS ARE UNDEFINED:
  sum_length
  permutation
  lequal

rule +25
n:natnum,x:list,y:list
#1 permutation(x,y)
|- permutation(cons(n,x),cons(n,y))=>TRUE;
 +25 ACCEPTED


rule +26
m:natnum,n:natnum,l:list,x:list,y:list
|- permutation(cons(m,l),cons(n,append(x,cons(m,y))))=>permutation(l,cons(n
    ,append(x,y)));
```

```
 +26 ACCEPTED


prove +15!
PROVING +15
x:cons,y:cons
|- permutation(merge_cons(x,y),append(x,y))=>TRUE

CURRENT GOAL:

[1]
x:cons,y:cons
|- permutation(merge_cons(x,y),append(x,y))=TRUE

     SIMPLIFIES TO:

x.cdr:list,x.car:natnum,x:cons,y.cdr:list,y.car:natnum,y:cons
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.cdr:list,x.car:natnum,x:cons,y.cdr:list,y.car:natnum,y:cons
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))


induct sum_length(cons(x.car,x.cdr),cons(y.car,y.cdr))!

NEW GOALS:

[1]
x.cdr:list,x.car:natnum,x:cons,y.cdr:list,y.car:natnum,y:cons
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) < sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))

     SIMPLIFIES TO:

x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) < sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) < sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))


type !equal(x.car,y.car)!

NEW GOALS:
```

```
[1]
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 lequal(x.car,y.car) : TRUE
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))

     SIMPLIFIES TO:

x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(cons(x.car,merge(x.cdr,cons(y.car,y.cdr))),cons(x.car,append(
    x.cdr,cons(y.car,y.cdr))))

[2]
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 lequal(x.car,y.car) : FALSE
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(merge_cons(cons(x.car,x.cdr),cons(y.car,y.cdr)),cons(x.car,append
    (x.cdr,cons(y.car,y.cdr))))

     SIMPLIFIES TO:

x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 not lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(list case of y.cdr | NIL: cons(x.car,x.cdr)  cons: merge_cons(
    cons(x.car,x.cdr),cons(y.cdr.car,y.cdr.cdr))|,cons(x.car,append(x.cdr,y.cdr)
    ))
2 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(cons(x.car,merge(x.cdr,cons(y.car,y.cdr))),cons(x.car,append(
    x.cdr,cons(y.car,y.cdr))))


type x.cdr!

NEW GOALS:

[1]
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
```

```
#1 x.cdr : NIL
#2 lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(cons(x.car,merge(x.cdr,cons(y.car,y.cdr))),cons(x.car,append(
    x.cdr,cons(y.car,y.cdr))))

     SIMPLIFIES TO:

TRUE

[2]
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 x.cdr : cons
#2 lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(cons(x.car,merge(x.cdr,cons(y.car,y.cdr))),cons(x.car,append(
    x.cdr,cons(y.car,y.cdr))))

     SIMPLIFIES TO:

TRUE
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 not lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(list case of y.cdr | NIL: cons(x.car,x.cdr)  cons: merge_cons(
    cons(x.car,x.cdr),cons(y.cdr.car,y.cdr.cdr))],cons(x.car,append(x.cdr,y.cdr)
    ))


type y.cdr|

NEW GOALS:

[1]
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
#1 y.cdr : NIL
#2 not lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
       x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
       x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(list case of y.cdr | NIL: cons(x.car,x.cdr)  cons: merge_cons(
    cons(x.car,x.cdr),cons(y.cdr.car,y.cdr.cdr))],cons(x.car,append(x.cdr,y.cdr)
    ))

     SIMPLIFIES TO:

TRUE

[2]
x.cdr:list,x.car:natnum,y.cdr:list,y.car:natnum
```

```
#1 y.cdr : cons
#2 not lequal(x.car,y.car)
&1 [x.cdr':list,x.car':natnum,x':cons,y.cdr':list,y.car':natnum,y':cons
   #1 sum_length(cons(x.car',x.cdr'),cons(y.car',y.cdr')) c sum_length(cons(
      x.car,x.cdr),cons(y.car,y.cdr))
   |- permutation(merge_cons(cons(x.car',x.cdr'),cons(y.car',y.cdr')),cons(
      x.car',append(x.cdr',cons(y.car',y.cdr'))))=>TRUE]
|- permutation(list case of y.cdr | NIL: cons(x.car,x.cdr)  cons: merge_cons(
    cons(x.car,x.cdr),cons(y.cdr.car,y.cdr.cdr))|,cons(x.car,append(x.cdr,y.cdr)
    ))

     SIMPLIFIES TO:

TRUE
PROOF OF +15 RELATIVE TO THE ASSERTIONS:
 +10 +20 +17 +7 +25 +19 +26 +21
UNPROVED RULES:  +26 +25 +23 +21 +20 +19 +17 +16 +14 +12 +11 +10 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;
prove +10;
PROVING +10
x:list
|- append(x,NIL)=>x


induct x!

NEW GOALS:

[1]
x:list
#1 x : NIL
&1 [x':list
   #1 x' c x
   |- append(x',NIL)=x' s> TRUE]
|- append(x,NIL)=x

     SIMPLIFIES TO:

TRUE

[2]
x:list
#1 x : cons
&1 [x':list
   #1 x' c x
   |- append(x',NIL)=x' s> TRUE]
|- append(x,NIL)=x

     SIMPLIFIES TO:

TRUE
PROOF OF +10 RELATIVE TO THE ASSERTIONS:
 +17
UNPROVED RULES:  +26 +25 +23 +21 +20 +19 +17 +16 +14 +12 +11 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

prove +14;
PROVING +14
x:list,y:list,z:list_of_cons
|- append(x,append(y,list_append(z)))=>append(append(x,y),list_append(z))


induct x!
```

```
NEW GOALS:

[1]
x:list,y:list,z:list_of_cons
#1 x : NIL
&1 [x':list,y':list,z':list_of_cons
   #1 x' c x
   |- append(x',append(y',list_append(z')))=append(append(x',y'),list_append(z')
      ) s> TRUE]
|- append(x,append(y,list_append(z)))=append(append(x,y),list_append(z))

     SIMPLIFIES TO:

TRUE

[2]
x:list,y:list,z:list_of_cons
#1 x : cons
&1 [x':list,y':list,z':list_of_cons
   #1 x' c x
   |- append(x',append(y',list_append(z')))=append(append(x',y'),list_append(z')
      ) s> TRUE]
|- append(x,append(y,list_append(z)))=append(append(x,y),list_append(z))

     SIMPLIFIES TO:

TRUE
PROOF OF +14 RELATIVE TO THE ASSERTIONS:
 +17
UNPROVED RULES:  +26 +25 +23 +21 +20 +19 +17 +16 +12 +11 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

prove +11;
PROVING +11
x:list
|- list_append(drop(x))=>x


induct x!

NEW GOALS:

[1]
x:list
#1 x : NIL
&1 [x':list
   #1 x' c x
   |- list_append(drop(x'))=x' s> TRUE]
|- list_append(drop(x))=x

     SIMPLIFIES TO:

TRUE

[2]
x:list
#1 x : cons
&1 [x':list
   #1 x' c x
   |- list_append(drop(x'))=x' s> TRUE]
|- list_append(drop(x))=x

     SIMPLIFIES TO:
```

```
TRUE
PROOF OF +11 RELATIVE TO THE ASSERTIONS:
 +17
UNPROVED RULES:  +26 +25 +23 +21 +20 +19 +17 +16 +12 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

prove +17!
PROVING +17
x:list
|- append(NIL,x)=>x

CURRENT GOAL:

[1]
x:list
|- append(NIL,x)=x

     SIMPLIFIES TO:

TRUE
PROOF OF +17
UNPROVED RULES:  +26 +25 +23 +21 +20 +19 +16 +12 +7
UNPROVED SYNTAX LEMMAS:  -16 -11 -10 -2

qed;

prove -11;
PROVING -11
l1:list,l2:list
|- append(l1,l2) : list


induct l1!

NEW GOALS:

[1]
l1:list,l2:list
#1 l1 : NIL
&1 [l1':list,l2':list
   #1 l1' c l1
   |- append(l1',l2') : list]
|- append(l1,l2) : list

     SIMPLIFIES TO:

TRUE

[2]
l1:list,l2:list
#1 l1 : cons
&1 [l1':list,l2':list
   #1 l1' c l1
   |- append(l1',l2') : list]
|- append(l1,l2) : list

     SIMPLIFIES TO:

TRUE
PROOF OF -11
UNPROVED RULES:  +26 +25 +23 +21 +20 +19 +16 +12 +7
UNPROVED SYNTAX LEMMAS:  -16 -10 -2

qed;
```

A 1.4 Example 4: McCarthy-Painter Compiler for Expressions

```
program mcp;

type locname = atom U natnum
type location = location(loc: locname, locval: integer)
type state = NIL U locations(first_loc: location, other_locs: state)
type load = load(loc_adr: locname)
type sto = sto(sto_adr: natnum)
type li = li(arg: integer)
type add = add(add_adr: locname)
type instr = load U sto U li U add
type code = NIL U instrs(first_instr: instr, other_instrs: code)

type expr = integer U atom U sum(expr1: expr, expr2: expr)

function append(x:code, y:code): code =
  code case x of
    NIL: y
    instrs: instrs(first_instr(x),append(other_instrs(x),y))

declare function plus(x:integer, y:integer): integer

function contents(l:locname,s:state): integer =
  state case s of
    NIL: ZERO
    locations: if l equals loc(first_loc(s)) then locval(first_loc(s))
               else contents(l,other_locs(s))

function update(ln:locname, lv:integer, s:state): state =
  state case s of
    NIL: locations(location(ln,lv),NIL)
    locations: if loc(first_loc(s)) equals ln then
                 locations(location(ln,lv),other_locs(s))
               else locations(first_loc(s),update(ln,lv,other_locs(s)))

function step(i:instr, s:state): state =
  instr case i of
    load: update(ZERO,contents(loc_adr(i),s),s)
    sto: update(sto_adr(i),contents(ZERO,s),s)
    li: update(ZERO,arg(i),s)
    add: update(ZERO,plus(contents(add_adr(i),s),contents(ZERO,s)),s)

function outcome(c:code, s:state): state =
  code case c of
    NIL: s
    instrs: outcome(other_instrs(c),step(first_instr(c),s))

function compile(tcount:suc, e:expr): code =
  expr case e of
    integer: instrs(li(e),NIL)
    atom: instrs(load(e),NIL)
    sum: append(append(compile(tcount,expr1(e)),
           instrs(sto(tcount),compile(suc(tcount),expr2(e)))),
         instrs(add(tcount),NIL))

function value(e:expr, s:state): integer =
  expr case e of
    integer: e
    atom: contents(e,s)
    sum: plus(value(expr1(e),s),value(expr2(e),s))

THE FOLLOWING FUNCTIONS ARE UNDEFINED:
  plus
THE FOLLOWING SYNTAX LEMMAS HAVE BEEN GENERATED BY THE PARSER:
```

```
[-1]
x:code,y:code
|- append(x,y) : code;

[-2]
x:integer,y:integer
|- plus(x,y) : integer;

[-3]
l:locname,s:state
|- contents(l,s) : integer;

[-4]
ln:locname,iv:integer,s:state
|- update(ln,iv,s) : state;

[-5]
i:instr,s:state
|- step(i,s) : state;

[-6]
c:code,s:state
|- outcome(c,s) : state;

[-7]
tcount:suc,e:expr
|- compile(tcount,e) : code;

[-8]
e:expr,s:state
|- value(e,s) : integer;


theorem e1
n:suc,e:expr,s:state
|- contents(ZERO,outcome(compile(n,e),s))=value(e,s);

THEOREM 1 ACCEPTED


rule +1
l:locname,n:integer,s:state
|- contents(l,update(l,n,s))=>n;

 +1 ACCEPTED


rule +2
c1:code,c2:code,s:state
|- outcome(append(c1,c2),s)=>outcome(c2,outcome(c1,s));

 +2 ACCEPTED


rule +3
n:suc,e:expr,s:state
|- contents(n,outcome(compile(suc(n),e),s))=>contents(n,s);

 +3 ACCEPTED


rule +4
e:expr,c:code,s:state
|- value(e,outcome(c,s))=>value(e,s);

 +4 ACCEPTED
```

```
rule +5
e:expr,n:natnum,v:integer,s:state
|- value(e,update(n,v,s))=>value(e,s);

 +5 ACCEPTED


prove e1;

PROVING e1
n:suc,e:expr,s:state
|- contents(ZERO,outcome(compile(n,e),s))=value(e,s)


induct e +!


NEW GOALS:

[1]
n:suc,e:expr,s:state
#1 e : integer
&1 [n':suc,e':expr,s':state
   #1 e' c e
   |- contents(ZERO,outcome(compile(n',e'),s'))=>value(e',s')]
|- contents(ZERO,outcome(compile(n,e),s))=value(e,s)

     SIMPLIFIES TO:

TRUE

[2]
n:suc,e:expr,s:state
#1 e : atom
&1 [n':suc,e':expr,s':state
   #1 e' c e
   |- contents(ZERO,outcome(compile(n',e'),s'))=>value(e',s')]
|- contents(ZERO,outcome(compile(n,e),s))=value(e,s)

     SIMPLIFIES TO:

TRUE

[3]
n:suc,e:expr,s:state
#1 e : sum
&1 [n':suc,e':expr,s':state
   #1 e' c e
   |- contents(ZERO,outcome(compile(n',e'),s'))=>value(e',s')]
|- contents(ZERO,outcome(compile(n,e),s))=value(e,s)

     SIMPLIFIES TO:

TRUE
PROOF OF e1 RELATIVE TO THE ASSERTIONS:
 +1 +2 +3 +5 +4
UNPROVED RULES:  +5 +4 +3 +2 +1
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


rule +6
n1:suc,n2:suc,e:expr,s:state
#1 n1 c n2
|- contents(n1,outcome(compile(n2,e),s))=>contents(n1,s);
```

```
 +6 ACCEPTED


rule +7
l1:locname,l2:locname,n:integer,s:state
#1 l1≠l2
|- contents(l1,update(l2,n,s))=>contents(l1,s);

 +7 ACCEPTED


prove +3|

PROVING +3
n:suc,e:expr,s:state
|- contents(n,outcome(compile(suc(n),e),s))=>contents(n,s)

CURRENT GOAL:

[1]
n:suc,e:expr,s:state
|- contents(n,outcome(compile(suc(n),e),s))=contents(n,s)

     SIMPLIFIES TO:

TRUE
PROOF OF +3 RELATIVE TO THE ASSERTIONS:
 +6
UNPROVED RULES:  +6 +5 +4 +2 +1
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


prove +6;

PROVING +6
n1:suc,n2:suc,e:expr,s:state
#1 n1 c n2
|- contents(n1,outcome(compile(n2,e),s))=>contents(n1,s)


induct e +|


NEW GOALS:

[1]
n1:suc,n2:suc,e:expr,s:state
#1 e : integer
#2 n1 c n2
&1 [n1':suc,n2':suc,e':expr,s':state
   #1 e' c e
   #2 n1' c n2'
   |- contents(n1',outcome(compile(n2',e'),s'))=>contents(n1',s')]
|- contents(n1,outcome(compile(n2,e),s))=contents(n1,s)

     SIMPLIFIES TO:


TRUE

[2]
n1:suc,n2:suc,e:expr,s:state
#1 e : atom
#2 n1 c n2
```

```
&1 [n1':suc,n2':suc,e':expr,s':state
   #1 e' c e
   #2 n1' c n2'
   |- contents(n1',outcome(compile(n2',e'),s'))=>contents(n1',s')]
|- contents(n1,outcome(compile(n2,e),s))=contents(n1,s)

     SIMPLIFIES TO:

TRUE

[3]
n1:suc,n2:suc,e:expr,s:state
#1 e : sum
#2 n1 c n2
&1 [n1':suc,n2':suc,e':expr,s':state
   #1 e' c e
   #2 n1' c n2'
   |- contents(n1',outcome(compile(n2',e'),s'))=>contents(n1',s')]
|- contents(n1,outcome(compile(n2,e),s))=contents(n1,s)

     SIMPLIFIES TO:

TRUE
PROOF OF +6 RELATIVE TO THE ASSERTIONS:
 +7 +2 +1
UNPROVED RULES:  +7 +5 +4 +2 +1
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;
prove +1;

PROVING +1
l:locname,n:integer,s:state
|- contents(l,update(l,n,s))=>n


induct s +1


NEW GOALS:

[1]
l:locname,n:integer,s:state
#1 s : NIL
&1 [l':locname,n':integer,s':state
   #1 s' c s
   |- contents(l',update(l',n',s'))=>n']
|- contents(l,update(l,n,s))=n

     SIMPLIFIES TO:

TRUE

[2]
l:locname,n:integer,s:state
#1 s : locations
&1 [l':locname,n':integer,s':state
   #1 s' c s
   |- contents(l',update(l',n',s'))=>n']
|- contents(l,update(l,n,s))=n

     SIMPLIFIES TO:

s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l:locname,n:integer
&1 [l':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
```

```
       s.other_locs)
   |- contents(l',update(l',n',s'))=>n']
|- if l equals loc(first_loc(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l then locations
    (location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     else contents(l,other_locs(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))=n
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l:locname,n:integer
&1 [l':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   |- contents(l',update(l',n',s'))=>n']
|- if l equals loc(first_loc(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l then locations
    (location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     else contents(l,other_locs(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))=n


type s.first_loc.loc equals ll


NEW GOALS:

[1]
s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l:locname,n:integer
#1 [s.first_loc.loc equals l] : TRUE
&1 [l':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   |- contents(l',update(l',n',s'))=>n']
|- if l equals loc(first_loc(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l then locations
    (location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     else contents(l,other_locs(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))=n

    SIMPLIFIES TO:

TRUE

[2]
s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l:locname,n:integer
#1 [s.first_loc.loc equals l] : FALSE
&1 [l':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   |- contents(l',update(l',n',s'))=>n']
|- if l equals loc(first_loc(if s.first_loc.loc equals l then
```

```
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l then locations
    (location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))
     else contents(l,other_locs(if s.first_loc.loc equals l then
    locations(location(l,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l,n,s.other_locs))))=n

     SIMPLIFIES TO:

TRUE
PROOF OF +1
UNPROVED RULES:  +7 +5 +4 +2
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


prove +2;

PROVING +2
c1:code,c2:code,s:state
|- outcome(append(c1,c2),s)=>outcome(c2,outcome(c1,s))


induct c1 +1


NEW GOALS:

[1]
c1:code,c2:code,s:state
#1 c1 : NIL
&1 [c1':code,c2':code,s':state
   #1 c1' c c1
   |- outcome(append(c1',c2'),s')=>outcome(c2',outcome(c1',s'))]
|- outcome(append(c1,c2),s)=outcome(c2,outcome(c1,s))

     SIMPLIFIES TO:

TRUE

[2]
c1:code,c2:code,s:state
#1 c1 : instrs
&1 [c1':code,c2':code,s':state
   #1 c1' c c1
   |- outcome(append(c1',c2'),s')=>outcome(c2',outcome(c1',s'))]
|- outcome(append(c1,c2),s)=outcome(c2,outcome(c1,s))

     SIMPLIFIES TO:

TRUE
PROOF OF +2
UNPROVED RULES:  +7 +5 +4
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


prove +4;

PROVING +4
e:expr,c:code,s:state
|- value(e,outcome(c,s))=>value(e,s)
```

```
induct c ⊣!


NEW GOALS:

[1]
e:expr,c:code,s:state
#1 c : NIL
&1 [e':expr,c':code,s':state
   #1 c' c c
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,outcome(c,s))=value(e,s)

     SIMPLIFIES TO:

TRUE

[2]
e:expr,c:code,s:state
#1 c : instrs
&1 [e':expr,c':code,s':state
   #1 c' c c
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,outcome(c,s))=value(e,s)

     SIMPLIFIES TO:

c.other_instrs:code,c.first_instr:instr,e:expr,s:state
&1 [e':expr,c':code,s':state
   #1 c' c instrs(c.first_instr,c.other_instrs)
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,step(c.first_instr,s))=value(e,s)
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
c.other_instrs:code,c.first_instr:instr,e:expr,s:state
&1 [e':expr,c':code,s':state
   #1 c' c instrs(c.first_instr,c.other_instrs)
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,step(c.first_instr,s))=value(e,s)


type c.first_instr!


NEW GOALS:

[1]
c.other_instrs:code,c.first_instr:instr,e:expr,s:state
#1 c.first_instr : load
&1 [e':expr,c':code,s':state
   #1 c' c instrs(c.first_instr,c.other_instrs)
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,step(c.first_instr,s))=value(e,s)

     SIMPLIFIES TO:

TRUE

[2]
c.other_instrs:code,c.first_instr:instr,e:expr,s:state
#1 c.first_instr : sto
&1 [e':expr,c':code,s':state
   #1 c' c instrs(c.first_instr,c.other_instrs)
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,step(c.first_instr,s))=value(e,s)
```

```
     SIMPLIFIES TO:

TRUE

[3]
c.other_instrs:code,c.first_instr:instr,e:expr,s:state
#1 c.first_instr : ii
&1 [e':expr,c':code,s':state
   #1 c' c instrs(c.first_instr,c.other_instrs)
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,step(c.first_instr,s))=value(e,s)

     SIMPLIFIES TO:

TRUE

[4]
c.other_instrs:code,c.first_instr:instr,e:expr,s:state
#1 c.first_instr : add
&1 [e':expr,c':code,s':state
   #1 c' c instrs(c.first_instr,c.other_instrs)
   |- value(e',outcome(c',s'))=>value(e',s')]
|- value(e,step(c.first_instr,s))=value(e,s)

     SIMPLIFIES TO:

TRUE
PROOF OF +4 RELATIVE TO THE ASSERTIONS:
 +5
UNPROVED RULES:  +7 +5
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


prove +5;

PROVING +5
e:expr,n:natnum,v:integer,s:state
|- value(e,update(n,v,s))=>value(e,s)


induct e +!


NEW GOALS:

[1]
e:expr,n:natnum,v:integer,s:state
#1 e : integer
&1 [e':expr,n':natnum,v':integer,s':state
   #1 e' c e
   |- value(e',update(n',v',s'))=>value(e',s')]
|- value(e,update(n,v,s))=value(e,s)

     SIMPLIFIES TO:

TRUE

[2]
e:expr,n:natnum,v:integer,s:state
#1 e : atom
&1 [e':expr,n':natnum,v':integer,s':state
   #1 e' c e
   |- value(e',update(n',v',s'))=>value(e',s')]
|- value(e,update(n,v,s))=value(e,s)
```

```
     SIMPLIFIES TO:

TRUE

[3]
e:expr,n:natnum,v:integer,s:state
#1 e : sum
&1 [e':expr,n':natnum,v':integer,s':state
   #1 e' c e
   |- value(e',update(n',v',s'))=>value(e',s')]
|- value(e,update(n,v,s))=value(e,s)

     SIMPLIFIES TO:

TRUE
PROOF OF +5 RELATIVE TO THE ASSERTIONS:
 +7
UNPROVED RULES:  +7
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


prove +7;

PROVING +7
l1:locname,l2:locname,n:integer,s:state
#1 l1≠l2
|- contents(l1,update(l2,n,s))=>contents(l1,s)


induct s +!


NEW GOALS:

[1]
l1:locname,l2:locname,n:integer,s:state
#1 s : NIL
#2 l1≠l2
&1 [l1':locname,l2':locname,n':integer,s':state
   #1 s' c s
   #2 l1'≠l2'
   |- contents(l1',update(l2',n',s'))=>contents(l1',s')]
|- contents(l1,update(l2,n,s))=contents(l1,s)

     SIMPLIFIES TO:

TRUE

[2]
l1:locname,l2:locname,n:integer,s:state
#1 s : locations
#2 l1≠l2
&1 [l1':locname,l2':locname,n':integer,s':state
   #1 s' c s
   #2 l1'≠l2'
   |- contents(l1',update(l2',n',s'))=>contents(l1',s')]
|- contents(l1,update(l2,n,s))=contents(l1,s)

     SIMPLIFIES TO:

s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l1:locname,l2:locname,n:integer
#1 l1≠l2
&1 [l1':locname,l2':locname,n':integer,s':state
```

```
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   #2 l1'≠l2'
   |- contents(l1',update(l2',n',s'))=>contents(l1',s')]
|- if l1 equals loc(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     else contents(l1,other_locs(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))=
    if l1 equals s.first_loc.loc then s.first_loc.locval else
    contents(l1,s.other_locs)
1 GOALS REMAIN TO BE PROVED

CURRENT GOAL:
s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l1:locname,l2:locname,n:integer
#1 l1≠l2
&1 [l1':locname,l2':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   #2 l1'≠l2'
   |- contents(l1',update(l2',n',s'))=>contents(l1',s')]
|- if l1 equals loc(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     else contents(l1,other_locs(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))=
    if l1 equals s.first_loc.loc then s.first_loc.locval else
    contents(l1,s.other_locs)


type s.first_loc.loc equals l2!


NEW GOALS:

[1]
s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l1:locname,l2:locname,n:integer
#1 (s.first_loc.loc equals l2) : TRUE
#2 l1≠l2
&1 [l1':locname,l2':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   #2 l1'≠l2'
   |- contents(l1',update(l2',n',s'))=>contents(l1',s')]
|- if l1 equals loc(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     else contents(l1,other_locs(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))=
    if l1 equals s.first_loc.loc then s.first_loc.locval else
    contents(l1,s.other_locs)

     SIMPLIFIES TO:
```

```
TRUE

[2]
s.other_locs:state,s.first_loc.locval:integer,s.first_loc.loc:locname
    ,l1:locname,l2:locname,n:integer
#1 [s.first_loc.loc equals l2] : FALSE
#2 l1≠l2
&1 [l1':locname,l2':locname,n':integer,s':state
   #1 s' c locations(location(s.first_loc.loc,s.first_loc.locval),
       s.other_locs)
   #2 l1'≠l2'
   |- contents(l1',update(l2',n',s'))=>contents(l1',s')]
|- if l1 equals loc(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     then locval(first_loc(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))
     else contents(l1,other_locs(if s.first_loc.loc equals l2 then
    locations(location(l2,n),s.other_locs) else locations(location(
    s.first_loc.loc,s.first_loc.locval),update(l2,n,s.other_locs))))=
    if l1 equals s.first_loc.loc then s.first_loc.locval else
    contents(l1,s.other_locs)

     SIMPLIFIES TO:

TRUE
PROOF OF +7
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2 -1

qed;


prove -1;

PROVING -1
x:code,y:code
|- append(x,y) : code


induct x!


NEW GOALS:

[1]
x:code,y:code
#1 x : NIL
&1 [x':code,y':code
   #1 x' c x
   |- append(x',y') : code]
|- append(x,y) : code

     SIMPLIFIES TO:


TRUE

[2]
x:code,y:code
#1 x : instrs
&1 [x':code,y':code
   #1 x' c x
   |- append(x',y') : code]
|- append(x,y) : code
```

```
     SIMPLIFIES TO:

TRUE
PROOF OF -1
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -3 -2

qed;


prove -3;

PROVING -3
l:locname,s:state
|- contents(l,s) : integer


induct s!


NEW GOALS:

[1]
l:locname,s:state
#1 s : NIL
&1 [l':locname,s':state
   #1 s' c s
   |- contents(l',s') : integer]
|- contents(l,s) : integer

     SIMPLIFIES TO:

TRUE

[2]
l:locname,s:state
#1 s : locations
&1 [l':locname,s':state
   #1 s' c s
   |- contents(l',s') : integer]
|- contents(l,s) : integer

     SIMPLIFIES TO:

TRUE
PROOF OF -3
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -4 -2

qed;


prove -4;

PROVING -4
ln:locname,lv:integer,s:state
|- update(ln,lv,s) : state


induct s!


NEW GOALS:

[1]
ln:locname,lv:integer,s:state
#1 s : NIL
&1 [ln':locname,lv':integer,s':state
   #1 s' c s
```

```
   |- update(ln',lv',s') : state)
|- update(ln,lv,s) : state

     SIMPLIFIES TO:

TRUE

[2]
ln:locname,lv:integer,s:state
#1 s : locations
&1 [ln':locname,lv':integer,s':state
   #1 s' c s
   |- update(ln',lv',s') : state]
|- update(ln,lv,s) : state

     SIMPLIFIES TO:

TRUE
PROOF OF -4
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -5 -2

qed;


prove -5;

PROVING -5
i:instr,s:state
|- step(i,s) : state


type !!


NEW GOALS:

[1]
i:instr,s:state
#1 i : load
|- step(i,s) : state

     SIMPLIFIES TO:

TRUE

[2]
i:instr,s:state
#1 i : sto
|- step(i,s) : state

     SIMPLIFIES TO:

TRUE

[3]
i:instr,s:state
#1 i : !!
|- step(i,s) : state

     SIMPLIFIES TO:

TRUE

[4]
i:instr,s:state
#1 i : add
|- step(i,s) : state
```

```
     SIMPLIFIES TO:

TRUE
PROOF OF -5
UNPROVED SYNTAX LEMMAS:  -8 -7 -6 -2

qed;


prove -6;

PROVING -6
c:code,s:state
|- outcome(c,s) : state


induct c!


NEW GOALS:

[1]
c:code,s:state
#1 c : NIL
&1 [c':code,s':state
   #1 c' c c
   |- outcome(c',s') : state]
|- outcome(c,s) : state

     SIMPLIFIES TO:

TRUE

[2]
c:code,s:state
#1 c : instrs
&1 [c':code,s':state
   #1 c' c c
   |- outcome(c',s') : state]
|- outcome(c,s) : state

     SIMPLIFIES TO:

TRUE
PROOF OF -6
UNPROVED SYNTAX LEMMAS:  -8 -7 -2

qed;


prove -7;

PROVING -7
tcount:suc,e:expr
|- compile(tcount,e) : code


induct e!


NEW GOALS:

[1]
tcount:suc,e:expr
#1 e : integer
&1 [tcount':suc,e':expr
```

```
   #1 e' c e
   |- compile(tcount',e') : code]
|- compile(tcount,e) : code

     SIMPLIFIES TO:

TRUE

[2]
tcount:suc,e:expr
#1 e : atom
&1 [tcount':suc,e':expr
   #1 e' c e
   |- compile(tcount',e') : code]
|- compile(tcount,e) : code

     SIMPLIFIES TO:

TRUE

[3]
tcount:suc,e:expr
#1 e : sum
&1 [tcount':suc,e':expr
   #1 e' c e
   |- compile(tcount',e') : code]
|- compile(tcount,e) : code

     SIMPLIFIES TO:

TRUE
PROOF OF -7
UNPROVED SYNTAX LEMMAS:  -8 -2

qed;


prove -8;

PROVING -8
e:expr,s:state
|- value(e,s) : integer


induct e!


NEW GOALS:

[1]
e:expr,s:state
#1 e : integer
&1 [e':expr,s':state
   #1 e' c e
   |- value(e',s') : integer]
|- value(e,s) : integer

     SIMPLIFIES TO:

TRUE

[2]
e:expr,s:state
#1 e : atom
&1 [e':expr,s':state
   #1 e' c e
   |- value(e',s') : integer]
```

```
|- value(e,s) : integer

     SIMPLIFIES TO:

TRUE

[3]
e:expr,s:state
#1 e : sum
&1 [e':expr,s':state
   #1 e' c e
   |- value(e',s') : integer]
|- value(e,s) : integer

     SIMPLIFIES TO:

TRUE
PROOF OF -8
UNPROVED SYNTAX LEMMAS:  -2

qed;
```

APPENDIX 2

TLV USER'S MANUAL

A 2.1 TLV Conventions

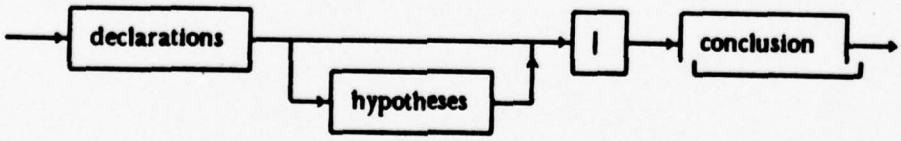
Before describing the commands accepted by the TYPED LISP Verifier, we need to define some terminology and notation. All statements in TLV are divided into four parts:

1. Variable declarations.
2. Quantifier-free hypotheses.
3. Induction hypotheses.
4. Conclusion.

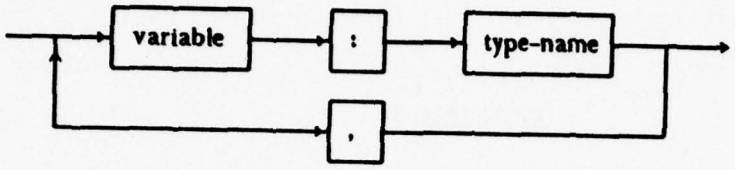
A variable declaration is simply a formula of the form $v : T$ where v is a variable and T is a type name. Every free variable appearing anywhere in a statement must be declared. The only statements which may contain induction hypotheses are the intermediate goals generated in the course of a proof. To permit easy reference to a some part of a statement, quantifier free hypotheses are labeled •1, •2, ... and induction hypotheses are labeled &1, &2, ... At any intermediate point in a proof in TLV, there is a list of goals which remain to be proved. These goals are labeled *1, *2, ... Goal *1 is called the current goal.

Statements serving as theorems or lemmas are called assertions. TLV assigns every assertion a unique assertion-name of the form •n, -n, or +n where n is a positive integer and the prefixes •, -, and + designate theorems, syntax lemmas, and user-specified lemmas, respectively. Assertions are parsed by the verifier according to the following syntax.

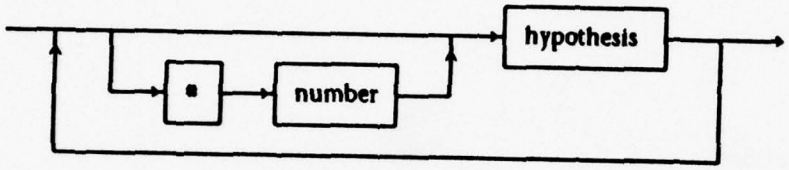
assertion



declarations

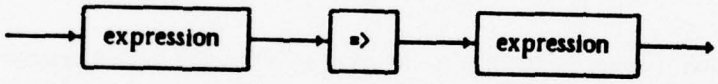


hypotheses

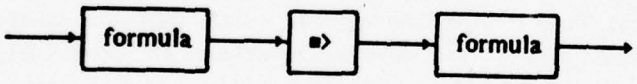


where each hypothesis is a formula and the conclusion is a formula rewrite pattern or an expression rewrite pattern with the syntax:

expression rewrite pattern

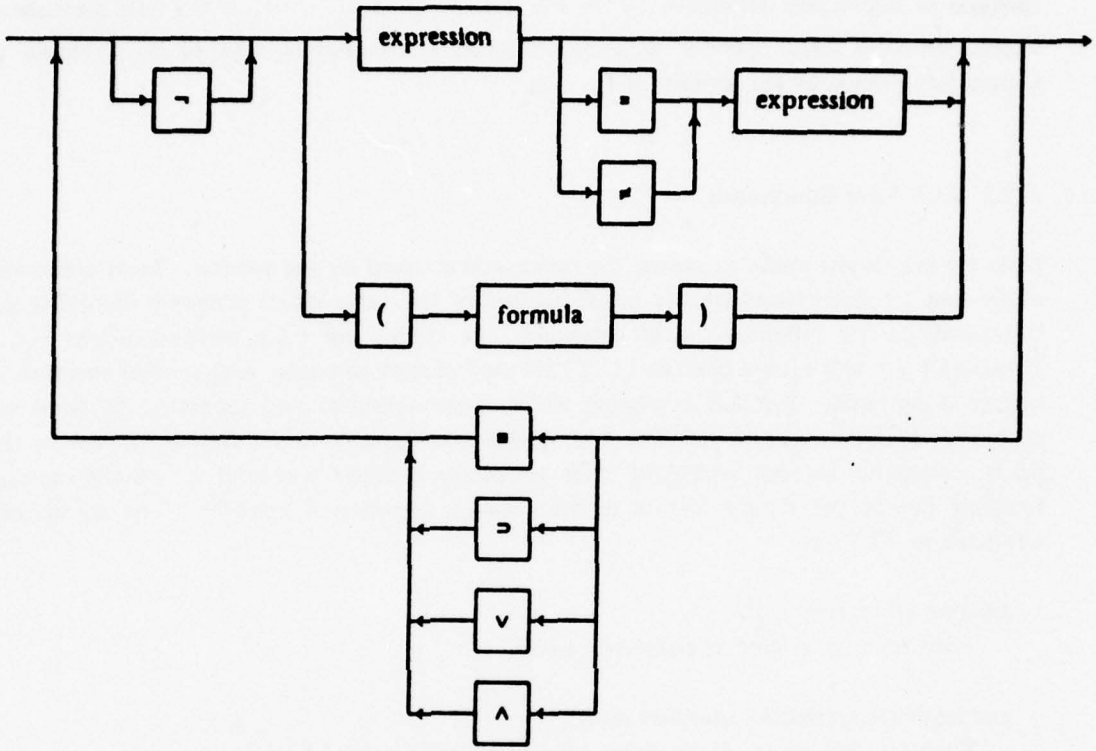


formula rewrite pattern



The command interpreter accepts labels of the form *#n* preceding hypotheses so it can parse assertions printed in the verifier's standard output format. They have no semantic significance.

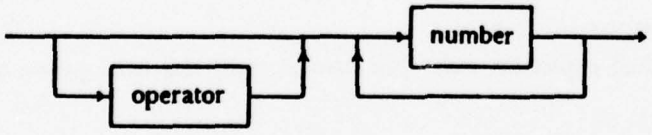
In TLV, formulas have the form:



where the precedence of the operators in decreasing order of binding power is ¬, ∧, ∨, ⊃, ■, and the binary operators are right associative.

TLV commands which take formulas or expressions as arguments also accept specifiers designating formulas or expressions in the current goal. Specifiers have the syntax:

specifier



A specifier composed solely of numbers designates the formula or expression within the current goal selected by the following rules. The first number of the sequence is a code indicating which part of the goal contains the specified formula or expression. Zero indicates the conclusion; a positive integer i indicates quantifier-free hypothesis $\bullet i$. Each subsequent number n (which must be positive) selects the nth operand of the currently selected formula or expression. The final formula or expression selected is the one designated by the specifier.

A more useful form for a specifier includes an operator op naming the head operator of the designated formula or expression (root of the standard syntax tree representation). The

formula or expression designated by the the specifier • *op* $n_1 \ldots n_k$ is the first formula or expression with head operator *op* encountered in a left-to-right scan of the formula or expression selected by the specifier • $n_1 \ldots n_k$.

A 2.2 TLV User Commands

Now we are finally ready to discuss the commands accepted by the verifier. Each command name may be abbreviated by any initial segment of the name which uniquely identifies the command. In the following list of commands, the symbol bar (|), pointy-brackets (<, >),braces ({, }), and square brackets ([,]) are used as meta-symbols. Any symbol enclosed in braces is optional. Symbols appearing within pointy-brackets and separated by bars are mutually exclusive alternatives. The bar symbol also appears as a terminal symbol in the prove command without ambiguity since no pointy brackets surround it. Finally, square brackets denote the Kleene closure of the enclosed sequence of symbols. The commands available in TLV are:

assume {*} *number* ⊲ | ⊳
Function: makes goal **number* into a rule.

consequence <*formula* | *specifier*> ⊲ | ⊳
Function: applies the consequence rule to goal *1 using the specified formula.

clear ⊲ | ⊳
Function: reinitializes the verifier.

delete [*hypothesis-number*] [<*induction-hypothesis-number*] ⊲ | ⊳
Function: applies the hypothesis deletion rule to the specified hypotheses.

disable [<*function-name* | *assertion-name*>] ⊲ | ⊳
Function: disables the specified expansion rules and lemmas until the next prove or enable command.

enable [<*function-name* | *assertion-name*>] ⊲ | ⊳
Function: enables the specified expansion rules and lemmas if they were disabled.

equals <{•} *number* | *formula* | *specifier*> <← | →> {*specifier*} ⊲ | ⊳
Function: applies the equals rule to the section of goal *1 indicated by *specifier* (if omitted the entire goal is specified) using equality-hypothesis •*number* or the new hypothesis specified. The latter option also creates the extra goal of proving the *general-formula* is a consequence of the goal hypotheses.

formula ⟨*formula* | *specifier*⟩ ⟨; | !⟩
Function: performs a formula split on goal *1 using the specified formula.

induct ⟨*expression* | *specifier*⟩ {⟨← | →⟩} ⟨; | !⟩
Function: applies the induction rule to goal *1 using the specified expression as the induction term. The optional pattern-specifier (the symbol ← or →) makes the induction-hypothesis into an expression rule or formula rule directed in the specified direction, if possible.

instantiate ⟨*assertion-name* | & *number*⟩
[*expression*]
⟨; | !⟩
Function: applies the instantiation rule to goal *1 using the specified assertion and the terms provided in the *expression-list*. Note: the *expressions* in the *expression-list* and the final terminator (the symbol ; or !) are entered in response to queries by the verifier.

list ⟨*assertion-class* | *assertion-name*⟩ ⟨; | !⟩
Function: lists all assertions in the specified *assertion-class* or the single assertion specified by *assertion-name* on the user's terminal.

occurs ⟨{*} *number* | *expression* | *specifier*⟩ ⟨{*} *number* | *expression* | *specifier*⟩ ⟨; | !⟩
Function: applies the occurs rule to the two expressions indicated. The first expression must be of the form $t_1 \subset t_2$ and the second of the form $t_2 \subset t_3$ where t_1, t_2, t_3 are terms.

program *file-name* ⟨; | !⟩
Function: reads the program from the file *file-name*.pg Note: the effect of reading several programs without performing an intervening **clear** command is cumulative. No previous definitions are destroyed.

prove *assertion-name* { | [*assertion-name*] } ⟨; | !⟩
Function: initializes the goal-list to the single goal specified by *assertion-name* and disables the rules specified by the *assertion-names* following the bar symbol (|). The verifier automatically disables any rules depending on the selected assertion or a disabled rule.

read *file-name* ⟨; | !⟩
Function: reads commands from the file *file-name*.pf until an end-of-file or error-condition is encountered.

replace *expression* {**by**} *variable-name* : *type-name* ⟨; | !⟩
Function: applies the replacement command to goal *1, replacing the specified

expression by *variable-name*.

rule *assertion* <; | !>

Function: creates a rule (lemma). An *assertion* with an expression rewrite pattern or a formula rewrite pattern for a conclusion creates the specified expression or formula rule. Similary, an *assertion* with a type expression for a conclusion creates the specified type rule. An *assertion* with a conclusion of the form $t_1 = t_2$ or $t_2 = t_1$ where t_1, t_2 are expressions and t_2 is composed solely from constructors and constants creates an expression rule with conclusion $t_1 \Rightarrow t_2$. An *assertion* with conclusion τ where τ is an expression (abbreviating a formula) creates the expression rule with conclusion $t_1 \Rightarrow$ **TRUE**. Any other *assertion* with conclusion α creates the formula rule with conclusion $\alpha \Rightarrow$ **TRUE**.

simplify {•} <*number*> <; | !>

Function: simplifies only the hypothesis •*number* within goal *1.

show {*number*} <; | !>

Function: list the last *number* proof steps on the user's terminal. If *number* is ommitted, all steps are listed.

status <; | !>

Function: lists the status of all assertions (theorems, lemmas syntax lemmas, expression-rules, type-rules, and formula-rules) known to the verifier.

undo {*number*} <; | !>

Function: undoes all proof steps back to step *number*. If *number* is omitted, one step is undone.

write {*file-name*} <; | !>

Function: writes all proof steps executed (but not undone) since the verifier was last cleared.

!

Function: simplifies goal *1.

A 2.3 Running TLV

The terminating symbol ; or ! tells the verifier whether or not to simplify the new goals generated (or if no new goals are generated goal *1). The symbol ! indicates that

simplification should be performed; the symbol ; indicates that simplification should be suppressed. However, simplification is never done after commands which do not generate proof steps: **clear, list, read, show, status, undo,** and **write.**

To start the TYPED LISP Verifier (TLV), the user must call the function RESET (with no arguments) from the top level of LISP, i.e. type

(RESET)

In response, the verifier will return the prompt symbol •, indicating it is waiting for a command. To return to the top level of LISP, the user simply types the symbol ∞ followed by a carriage return. To reenter the verifier from the top level of LISP after exiting or encountering a system error, the user simply calls the function REE (with no arguments). Unlike RESET, REE does not destroy the previous state of the verifier.

A 2.4 TYPED LISP Syntax Error Messages

2.4.1. Minor Errors

The parser continues parsing normally despite the ocurrence of errors from the list below.

ERROR NUMBER	EXPLANATION
-1	The previous character in the input stream is invalid. The parser skips and ignores the character.
-2	The actual type of the preceding *expression* does not intersect its allowed type.
-3	In an enumeration, the preceding constant has already appeared in the list-of-constants.
-4	The function-name specified in the preceding function-declaration has already been declared earlier in the program.
-5	The current function-definition conflicts with the declaration of the same function-name earlier in the program.

2.4.2. Major Errors

The parser recovers from the following errors by skipping the remainder of the definition or assertion containing the error and resuming parsing at the beginning of the next definition or command in the input stream.

ERROR NUMBER	EXPLANATION
2	A type-name must head a case-alternative.
3	An incorrect type-name heads the current case-alternative.
4	The symbol : must follow the type-name heading a case-alternative.
5	In a case-expression, case must follow the type-name heading the expression.

simplification should be performed; the symbol ; indicates that simplification should be suppressed. However, simplification is never done after commands which do not generate proof steps: **clear, list, read, show, status, undo,** and **write.**

To start the TYPED LISP Verifier (TLV), the user must call the function RESET (with no arguments) from the top level of LISP, i.e. type

(RESET)

In response, the verifier will return the prompt symbol •, indicating it is waiting for a command. To return to the top level of LISP, the user simply types the symbol ∞ followed by a carriage return. To reenter the verifier from the top level of LISP after exiting or encountering a system error, the user simply calls the function REE (with no arguments). Unlike RESET, REE does not destroy the previous state of the verifier.

6	In a case-expression, of must follow the index expression.
7	In a disjoint-union, more than one component type-name must appear.
8	In an if-expression, then must follow the boolean expression.
9	In an if-expression, else must follow the consequent expression.
10	In a function-call, the symbol) or , must follow an argument.
11	In the preceding function-call, there are too few arguments.
12	A term must begin with an identifier, a constant, or the symbol [.
13	In the current expression, the preceding variable-name is not defined.
14	In the current expression, the preceding identifier is not defined.
15	In the a function-call, the symbol (must follow the function-name.
16	In the preceding function-call, there are too many arguments.
17	At the beginning of a definition, either type, function, or declare must appear.
18	In a constructor definition, a selector-name must head a selector-field.
19	The preceding identifier heading a selector-field has already been defined.
20	In a constructor-definition, the symbol : must follow a selector-name.
21	In a selector-field, a type-name must follow the symbol : .
22	The preceding type-name is undefined. Forward type references are permitted only in selector-fields within recursive-unions.
23	The parser was forced to abandon translatingthe current case-expression because of a syntax error in the definition of the case-type-name.

24 In a constructor-definition, the symbol (must follow the constructor-name.

25 In a constructor-definition, the symbol) must follow the selector-field-list.

26 In a type-definition, an identifier must follow type.

27 In the current type-definition, the preceding identifier has already been defined.

28 In a type-definition, the symbol = must follow the defined-type-name.

29 In an enumeration, a constant must follow the symbol { or , .

30 In an enumeration, the symbol { must follow the list-of-constants.

31 In a type-definition, the symbol { or a type-name must follow the symbol =.

32 In the current type-definition, the preceding type-name is undefined.

33 In the current disjoint-union or recursive-union, the preceding type intersects another type in the union.

35 In the constructor-definition-list within a recursive-union, an undefined identifier serving as a constructor-name must follow the symbol U.

36 A bracketed-expression must end with the symbol] .

37 In a function-definition, an undefined identifier must follow function.

39 In a function-definition, the symbol (must follow the function-name.

40 In a parameter-list within a definition or assertion, a variable-name must appear at the beginning of the list and following each , .

42 In the current parameter-list, the preceding variable-name appears twice.

43 In a parameter-list, the symbol : must follow a variable-name.

44 In a parameter-list, a type-name must follow the symbol : .

46 In a function-definition or function-declaration, the symbol (must follow the parameter-list.

47 In a function-definition or function-declaration, the symbol : must follow function (parameter-list) : .

48 In a function-definition or function-declaration, a type-name must follow function type-name (parameter-list) : .

49 In a function-definition, the symbol = must precede the the expression forming the body of the function.

50 The current function-definition conflicts with the declaration of the same function-name earlier in the program.

51 In a function-declaration, function must follow declare.

52 In a type-test, a type-name must follow term : .

101 In a assertion or induction-hypothesis, the delimiter | must precede the consequent.

102 In an atomic-formula a type-name must follow the predicate symbol : .

103 The predicate symbol => must not appear in an assertion which is not a rule.

104 In an atomic-formula, the only infix-predicates permitted are: : ¬: = ≠ ∈ ¬∈ .

105 A parenthesized-formula must end with the symbol) .

6.5 TYPED LISP Verifier Command Errors

If the command parser detects an error while reading from the user terminal, it skips to first ; or ! following the error. If the parser finds an error in a proof file, it closes the file and selects the user terminal for input.

ERROR NUMBER	EXPLANATION
1	Illegal command name.
2	A number was expected, but not found.
3	The selected goal does not exist.
4	No goals exist; the preeding command is inapplicable.
5	The selected hypothesis is not an equality formula.
6	The preceding specifier does not identify an expression.
7	The preceding specifier does not identify an formula.
8	The selected hypothesis does not exist.
9	The selected hypothesis is not an implication.
10	The type of the induction expression is not recursive.
11	The selected assertion or induction hypothesis does not exist.
12	The selected assertion depends on the assertion being proved.
13	Illegal assertion-name.
14	The selected hypothesis is not an occurs formula.
15	The selected hypothesis is not an or formula.
16	A proof is in progress; the prove command is illegal.
17	The selected assertion has already been proved.
18	Illegal variable name.
19	The symbol : is expected following the variable-name in a replace command.
20	Illegal type-name.

21	The type of the specified expression cannot be split.
22	No proof steps exist.
23	The specified file-name is not a lower-case-identifier.
24	The symbol → or ← is missing in an equality command.
25	The symbol => is used illegally within a rule.
26	The type of the specified expression does not intersect the type of the instantiated variable.
27	The selected lemma cannot be instantiated because it contains induction hypotheses.
28	A program command in a command file generated a syntax error.

APPENDIX 3

CALL-BY-VALUE FIXED-POINTS

One of the most popular models of computation is the least fixed-point approach to semantics originated by Kleene [Kleene 1952] and refined by Scott [1970, 1971], Park [1969], Milner [1973], and others. Least fixed-point semantics interprets a function defined by a recursion equation as the least fixed-point of the the functional associated with the right side of the equation. In other words, if f is defined by the recursion equation:

$$f(x_1, ..., x_n) \equiv \tau(x_1, ..., x_n),$$

then f is interpreted as the least fixed-point of the functional $\tau^*[f]$ defined by

$$\tau^*[f](x_1, ..., x_n) = \tau(x_1, ..., x_n)$$

using the standard partial ordering on partial functions. Of course, suitable restriction must be placed on the domain of data values and on the base functions appearing in τ for the least fixed-point to exist [See Milner 1973, Manna 1975].

Unfortunately, the least fixed-point interpretation for a function defined by a recursion equation frequently differs from the standard call-by-value interpretation used in most programming languages. Some computer scientists have been so impressed by the elegance of the least fixed-point interpretation for recursive functions that they have called the call-by-value interpretation incorrect! My own view is that while least fixed-point semantics is mathematically elegant in many respects, it is not an appropriate model for functions defined by recursion in real programming languages. Call-by-value semantics is much more intuitively comprehensible; few programmers think of the recursive procedures they write as the least fixed-points of functionals. The most appealing feature of least fixed-point semantics is that it defines the meaning of recursive functions without resorting to defining an interpreter for recursion equations, the usual method for formally defining the call-by-value interpretation. What is not widely appreciated is that the least fixed-point approach to semantics can be used to define the call-by-value interpretation for a recursion equation in a simple, elegant manner. Using the tools of least fixed-point semantics, I will define what I call the *least call-by-value fixed-point* of a recursion equation.

First, we need to review the standard definitions of least fixed-point semantics:

Definition. A complete partial ordering is a pair $\langle D, \sqsubseteq \rangle$ where D is any domain and $\sqsubseteq$ is a partial ordering on D such that:

a. There is a least element $\omega \in D$, i.e. for all $x \in D$, $\omega \sqsubseteq x$.

b. Every ascending sequence $X = x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ has a least upper bound denoted l.u.b. X.

Definition. A complete partial ordering D under the relation $\sqsubseteq$ is a *flat domain* iff for all $x,y \in D$, $x \sqsubseteq y$ iff $x = y$ or $x = \omega$. (The data domains for all existing practical programming languages are flat.)

Definition. Let D_1 and D_2 be complete partial orderings. A function f mapping D_1 into D_2 is *montonic* iff for any two elements $x,y \in D_1$, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. The function f is *continous* iff for every ascending sequence $X = \{x_i \mid i = 1,2,\ldots\}$ in D_1, $f(\text{l.u.b } X) = \text{l.u.b } \{f(x_i) \mid i = 1,2,\ldots\}$.

Notation. Let D_1, D_2 be two complete partial orderings. We denote the set of continuous functions mapping D_1 into D_2 by $[D_1 \rightarrow D_2]$.

Theorem A. Let D be complete partial ordering under $\sqsubseteq$. The cartesian product D^n is a complete partial ordering under the relation $\sqsubseteq_n$ defined by:

$$[x_1, \ldots, x_n] \sqsubseteq [y_1, \ldots, y_n] \quad \text{iff} \quad x_i \sqsubseteq y_i \text{ for } i = 1, \ldots, n.$$

Proof. See [Manna 1974].

Theorem B. If D_1 and D_2 are complete partial orderings, then $[D_1 \rightarrow D_2]$ is a complete partial ordering under the relation $\sqsubseteq$ defined for $f,g \in [D_1 \rightarrow D_2]$ by:

$f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in D_1$

Proof. See [Milner 1973].

Theorem C. Let D be complete partial ordering under $\sqsubseteq$, and let f be a continuous function mapping D into D. The function f has a unique least fixed-point . (In fact, the theorem is true under the weaker assumption that f is monotonic, but the less general form of theorem given here is sufficient for most purposes including ours.)

Proof. See [Milner 1973].

Notation. Let τ be a term composed from monotonic functions; constants the variables x_1,

..., x_n; and the function symbol f. Given the interpretation $\underline{f} \in [D^n \to D]$ for f and the interpretations $\underline{x_i} \in D$ for x_i, $i = 1, ..., n$, we denote the corresponding interpretation of τ by:

$$\langle \tau; \underline{f}; \underline{x_1}, ..., \underline{x_n} \rangle$$

Theorem D.. Let τ be a term composed from monotonic functions; constants the variables $x_1, ..., x_n$; and the function symbol f. Let $\underline{f} \in [D^n \to D]$, and let $\underline{x_i} \in D$, $i = 1, ..., n$. The functional $\underline{\tau}$ defined by

$$\underline{\tau}[\underline{f}](\underline{x_1}, ..., \underline{x_n}) = \langle \tau; \underline{f}; \underline{x_1}, ..., \underline{x_n} \rangle$$

is continuous.

Proof. See [Manna 1974] for a sketchy proof.

Now we are finally ready to define the least call-by-value fixed-point of a singly recursive recursion equation. The least call-by-value fixed-points of a system of mutually recursive equations is a straightforward generalization.

Definition. Let D be a flat domain under the relation $\sqsubseteq$ with subsets (types) $T_1, T_2, ... T_n$ excluding the undefined object ($\underline{\omega}$). Let f be the function symbol defined by the recursion equation E:

$$f(x_1: T_1, ..., x_n: T_n) = \tau(x_1, ..., x_n)$$

where τ is a term constructed from monotonic functions on cartesian products of D, the variables $x_1, ..., x_n$, and the uninterpretated n-ary function symbol f. Let $\underline{\tau^*}$ denote the functional mapping $[D^n \to D]$ into $[D^n \to D]$ defined by:

$$\underline{\tau^*}[f](x_1, ..., x_n) = \langle \text{if } x_1: T_1 \wedge \ldots \wedge x_n: T_n \text{ then } \tau(x_1, ..., x_n) \text{ else } \omega ;\ \underline{f};\ \underline{x_1}, ..., \underline{x_n} \rangle .$$

where if-then-else denotes the standard conditional function, $:T_i$ is the strict characteristic function for type T_i, and $\wedge$ denotes the standard boolean function "and". Since all of these functions are monotonic, $\underline{\tau^*}$ is continuous by Theorem D. A *call-by-value fixed-point* of the recursion equation E is any fixed-point of the functional $\underline{\tau^*}$. A *least call-by-value*

fixed-point g of the recursion equation $\mathbf{E}$ is a call-by-value point of $\mathbf{E}$ which has the property $g \sqsubseteq h$ for all call-by-value fixed-points h of $\mathbf{E}$.

The least call-by-value fixed-point of a recursion equation obviously corresponds to the call-by-value interpretation for the function defined by the equation, since it always "evaluates" each argument to ascertain that it is defined and belongs to the proper type.